OMNIVIS: 3D Space and Camera Path Reconstruction for Omnidirectional Vision

Jose Luis Ramirez Herran

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

February 1, 2010

Abstract

In this project we address the problem of reconstructing a scene and the camera motion from the image sequence taken by an **Omnidirectional camera** in both semi-synthetic and real scenes.

Initially we proposed to use one of the open source feature detectors that were available, but we decided to create our own **feature detector** in C. The available detectors were implemented as prototypes and they only work with gray scale images. Implementing our own **feature detector** provided us with two main advantages: First, we based our tracking system in the faster possible detector and second, our **feature extraction** algorithms work with color pictures. The **feature tracking** was also implemented in C. Since the mathematical nature of this inverse problem is ill-posed, **linear least squares** methods are used to get an estimate of the structure of the scene and the camera motion. This estimate, called **3D reconstruction**, is the final step of the proposed pipeline and it is implemented in Mathematica with the advantage of using a powerful library of optimized existing routines. Most of the programming routines that we use are from linear algebra.

Keywords: **Computer vision**, **Omnidirectional vision**, **Omnidirectional video**, **Omnidirectional camera**, **Structure from motion**, **3D reconstruction**, **linear least squares** methods, **Corner detection**, **feature tracking**, **off-line processing**, **parabolic mirror**, **catadioptric sensor**.

Author's Biographical Sketch

Jose Luis Ramirez Herran has a bachelor degree in **Mechanical Engineering** from Universidad Nacional de Colombia. In Colombia, his country of origin, he was interested mainly in using **finite element analysis** for engineering design, and **robotics**. After graduating he started a company called Finitos Ltd. dedicated to offer analysis in engineering for industrial applications using the finite element approach. At the same time he participated as an author and organizer of the first conferences on finite elements and numerical analysis in Colombia in 1994. For 4 years, he was an **instructor** on introductory calculus and physics for college students. He was invited to present a paper at the 1998 ANSYS International Conference in Pittsburgh, PA. With the idea of studying in the U.S.A. he moved to Boston and taught **mathematics** for high school for 5 years while started taking classes at Harvard. The new skills on math, computer graphics and programming languages from the **master in IT** program allowed him to successfully change his career and become an IT professional in the real world. His main interests are **computer graphics**, **computer vision**, and **parallel computing** with the GPU. In the future he wants to get a bf PhD in Science, Technology and Management, and teach at the university level.

Dedication

To my family - my source of inspiration and strength.

Acknowledgments

Thanks to my thesis director, Oliver Knill, for years of consistent support, for having confidence in my abilities, and for sharing with me his wisdom, and his original strategies for problem solving. It has truly been a privilege to work with such an effective and nice person. Thanks to Henry Leitner and the Extension School for supporting the class Maths300R. Thanks Prof. Todd Zickler for letting me attend to his Computer vision class. Thanks to Jeff Parker for being so available and for his effective help in the whole process of the thesis. And thanks to everyone that collaborated with me in the editing of this document.

Contents

1 Introduction **1**

1.1 Prior work on SFM for Omnidirectional vision 7

1.2 Organization of this document 12

2 Image Processing and Computer Vision **15**

2.1 Image processing 16

2.2 Movies 16

2.3 Computer vision methods 16

2.4 Low-level vision techniques 17

2.4.1 Low-level vision techniques: Basic operations 18

2.4.2 Low-level vision: Edge and corner operations 21

2.5 Local edge detectors 22

2.6 Local corner detectors 24

2.6.1 Cross corner detector 24

2.6.2 Kitchen-Rosenfeld Corner Detection 24

2.6.3 Harris Corner Detection 28

2.6.4 Smith Univalue Segment Assimilation Nucleus 31

2.6.5 Algebraic corner detection 32

2.6.6 An application of Corner detection 33

2.7 Middle-level Vision: Structure from Motion 35

3 Omnidirectional cameras **37**

3.1 Man made panoramic vision 37

3.2 Omnidirectional Vision in Nature 40

3.3 Unwrapping an Omnidirectional image 40

3.4 POV-Ray's Omnidirectional Camera 42

3.4.1 Coordinate system 43

3.4.2 POV-Ray Camera 44

3.4.3 Camera Calibration in POV-Ray 45

4 Mathematical Model of the 3D Reconstruction **47**

4.1 Structure from motion 48

4.2 Mathematics of the reconstruction in the planar case 48

4.3 Mathematics of the reconstruction in the 3D case 54

4.4 Error Estimation 60

5 Tracking and 3D reconstruction **62**

5.1 The correspondence problem. 62

5.2 Tracking points in synthetic scenes. 64

5.2.1 KLT Corner Detector with tracking 65

5.2.2 OMNIVIS Corner Detector with tracking 67

5.3 Tracking strategy 68

5.4 Tracking corners in synthetic scenes 73

5.5 Tracking corners in real scenes 76

5.5.1 Florida movie 76

5.5.2 Harvard University Science Center movies 78

5.5.3 Rotating a Rubik cube movie 79

5.5.4 Ballons Movie 81

5.5.5 Robot taking a movie of a small scene 82

5.6 3D Reconstruction for Omnidirectional vision 82

5.6.1 3D Reconstruction in a synthetic scene 83

6 Software implementation 88

6.1 Functional Requirements 88

6.2 Non-functional Requirements 89

6.3 Development Environment 89

6.4 Programming Languages 90

6.5 Implementation 90

7 Summary and Conclusions 92

7.1 Difficulties of tracking 92

7.2 Difficulties of the reconstruction 93

7.3 Future work to do 93

7.4 Lessons learned 94

Appendices 101

A Input data for the reconstruction of the synthetic scene 101

B Image formats 104

C Source Code 107

List of Figures

1.1 The Italian Job 2

1.2 Google street view 3

1.3 Parabolic mirror 3

1.4 Omnidirectional camera 4

1.5 Image stitching 4

1.6 POV-Ray Camera Motion Path 6

1.7 POV-Ray Movie 6

1.8 PROFORMA 10

1.9 Mixing Catadioptric and Perspective Cameras 11

2.1 Thresholding 18

2.2 Image histogram 19

2.3 Image Brightness 20

2.4 Thresholding . . . 20

2.5 Gaussian operator . . . 21

2.6 Detecting horizontal crosses . . . 25

2.7 Gradient operator . . . 25

2.8 Gradient with smoothing . . . 27

2.9 Application of the Gradient . . . 28

2.10 Gradient and curvature . . . 28

2.11 SIFT . . . 31

2.12 Algebraic corner detection . . . 34

2.13 Gradient and Hamiltonian field . . . 35

2.14 Drawing along the Hamiltonian field . . . 36

3.1 Mirrors . . . 38

3.2 Polar coordinates . . . 39

3.3 Eyes of a fly . . . 40

3.4 Gigantocypris . . . 41

3.5 Unwrapping using polar coordinates . . . 42

3.6 POV-Ray Coordinate System . . . 43

3.7 POV-Ray Camera . . . 44

3.8 Calibrating POV-Ray - Sphere with spheres . . . 46

3.9 Calibrating POV-Ray camera . . . 46

4.1 General situation in the plane . . . 49

4.2 2D example of cameras and points . . . 50

4.3 Autocalibration . . . 52

4.4 Cameras and Points . . . 52

4.5 3D Situation . . . 55

4.6 3D Example . . . 57

4.7 3D Reconstruction . . . 58

5.1 Point correspondences in panoramic pictures . . . 63

5.2 Matching points . . . 65

5.3 KLT tracking . . . 66

5.4 KLT tracking paths . . . 67

5.5 Tracking points . . . 67

5.6 Tracking points . . . 69

5.7 Keyframes . . . 69

5.8 Coding the points tracked 70
5.9 Multiple Gaussian smoothing 70
5.10 Coding curvature information in images 72
5.11 Coding gradient information in images 72
5.12 Curvature threshold 72
5.13 Gradient threshold 72
5.14 POV-Ray scene 73
5.15 Tracking example - scene 1 74
5.16 Tracking example - paths scene 1 74
5.17 Tracking example - scene 2 75
5.18 Tracking example - paths scene 2 75
5.19 Map of Palm Beach, FL 76
5.20 Omnicamera mounted in a Subaru 77
5.21 Unwrapped panorama - Palm Beach, FL 77
5.22 Tracking corners - Palm Beach, FL 78
5.23 Tracking results - Florida 78
5.24 Measurements - Florida 79
5.25 Tracking corners in a corridor 80

5.26 Tracking corners - Harvard Math Dept. 80
5.27 Tracking corners in a rotating object 81
5.28 Tracking corners in balloons 81
5.29 Robot with Omnidirectional camera 82
5.30 Tracking corners on blocks 83
5.31 OMNIVIS's Pipeline 83
5.32 Scene to be reconstructed 84
5.33 Tracked points in the synthetic scene 84
5.34 Reconstruction of the synthetic scene 87

6.1 Ominvis pipeline 91

A.1 Tracked corners in the first frame 102
A.2 Tracked corners in the last frame 103

B.1 Example of PPM 106

List of Tables

1.1 3D reconstruction using different catadioptric systems 13

1.2 3D reconstruction using multiple camera systems 14

2.1 Computer vision methods 17

Chapter 1

Introduction

Reconstructing the 3D structure and camera motion from a sequence of pictures is a theoretically interesting problem with many practical applications such as robot visual navigation and other computer vision tasks. This problem is called **Structure from motion** and it is a central problem in Computer Vision which nowadays constitutes a relative new field of research.

In figure 1.1 we show a typical situation in which this kind of problem comes out. In the figure we summarize a scene in the movie "the Italian Job", where the computer genius "Napster" builds a 3D map of a building from a movie taken by a wearable camera. How does "Napster" make his computer create a 3D map of the building based exclusively in the sequence of pictures taken inside the building? A similar implementation of a program that could do not only such a map but also the trajectory of the camera is the focus of this thesis. In the same figure, we also show an outline of the reconstruction process summarized in two steps: First, a camera captures a sequence of pictures ($frames 1, 2, ..., i, ..., n$) and then a Computer program generates a three-dimensional map of the building.

This kind of approach is called **off-line process** since the processing does not start until all the pictures are captured and stored. Off-line process is a common term in the **Com-**

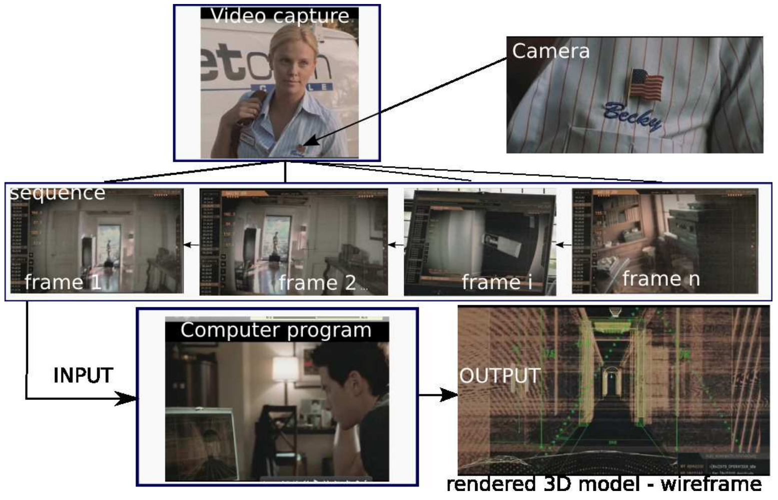


Figure 1.1: Creating a map of a building from video, in the Italian Job Movie - Paramount Pictures

puter vision literature and this is the condition under which most successful **structure from motion** work has been achieved. The other approach is called **online** or **real-time** process, in which the pictures are processed immediately after they are taken. For this thesis, we implement an off-line approach to perform a 3D reconstruction based on **structure from motion** algorithms. Our main focus is feature tracking, an important ingredient for the 3D reconstruction.

With our solution, it should be possible in principle to take the **Google street view** map 360 degree pictures of Boston and reconstruct a large part of the city as 3D. Nobody seems be able to do that in an automatic way. The potential of such a reconstruction would be immense. For example, it would be possible to build environments for games which are based on real world. In figure 1.2 we show the Memorial Hall at Harvard University and a 3D model made manually and included in a archive called the 3D Building warehouse which we captured by using **Google Earth**.

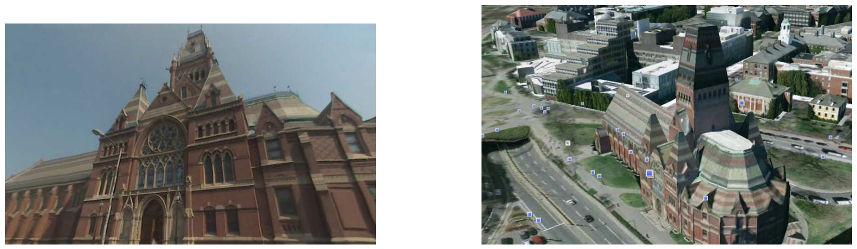


Figure 1.2: Google street view 360 degree panorama and 3D Building from Google Earth

The camera

A **catadioptric Omnidirectional vision sensor** is one where lens (*dioptrics*) and a curved mirror (*catoptric*) are combined to provide a 360 degrees view of the environment. In this project we use catadioptric Omnidirectional vision sensor which combines a point and shoot camera with a parabolic mirror as the one shown in figure 1.3.

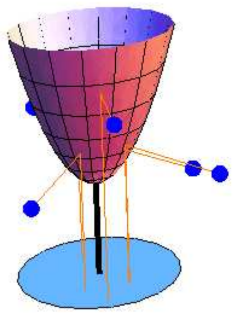


Figure 1.3: Parabolic mirror and a few rays projected to the camera

The pictures taken with **Omnidirectional camera**, as shown in figure 1.4, are easy to unwrap, have small distortion and a wide field of view. This **unwrapping** is a simple transformation from polar to rectangular coordinates.

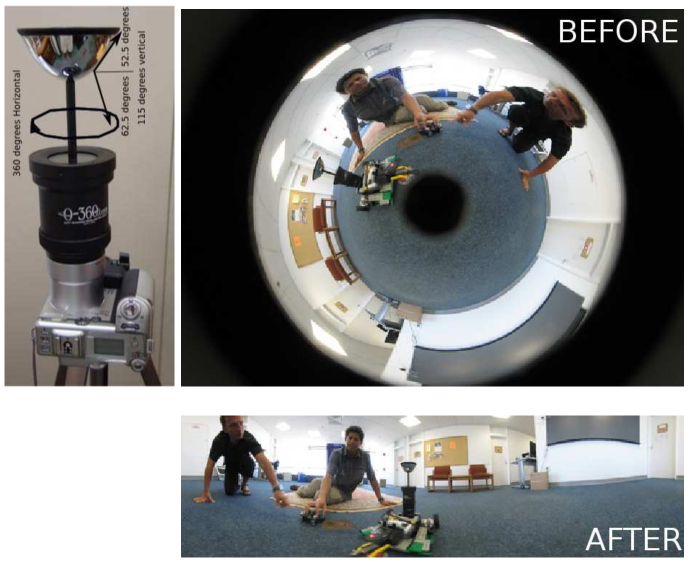


Figure 1.4: 0-360 camera system and Omnidirectional picture before and after unwrapping

After unwrapping, the Omnidirectional images are panoramas. Panoramas can also be created by stitching multiple perspective images together as we show in figure 1.5. For creating a panorama using the Omnidirectional camera we need only one shot.

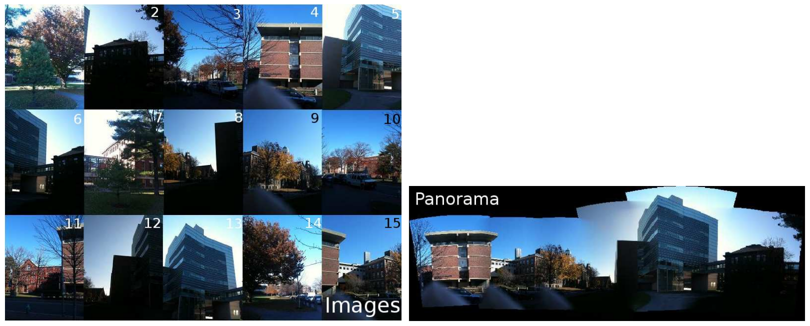


Figure 1.5: Panorama from image stitching using AutoStitch IPhone from Cloudburst Research Inc.

Mathematical approach

The mathematical approach used for solve the **structure from motion** for **Omnidirectional cameras** is based on the reduction of the visual perception problem to an optimization challenge. The definition of the objective function and the optimization process itself is ill-posed, since the number of variables to be recovered is much larger than the number of constraints. If a system of equations has more constraints than parameters, then a **least square approach** is needed. Selecting the most appropriate technique to address visual perception is rather task-driven and one cannot claim the existence of a universal solution to most of the visual perception problems [Paragios, 2006].

Synthetic problem

As an intermediate step we propose the following problem called the **semi synthetic reconstruction** problem. For solving this problem we build a synthetic city in the open source system called **Persistence Of Vision ray tracer** (POV-Ray), which is basically a scene made up of cubes of various sizes and textures. We move the built-in POV-Ray camera through this scene and make a movie. The sequence of pictures from the movie is the input to our reconstruction program. The next figure shows an example of a sequence of **panoramic images** that we will use, and the corresponding camera path.

We will get the positions and orientations of the cubes, extract the textures and rebuild a new scene in POV-Ray with these data. Then we record the same movie in the reconstructed world. This allows us to compare the quality of the reconstruction and see places where the reconstruction has ambiguities, a mathematical concept we have explored theoretically too in 2006 [Knill and Ramirez-Herran, 2007b].

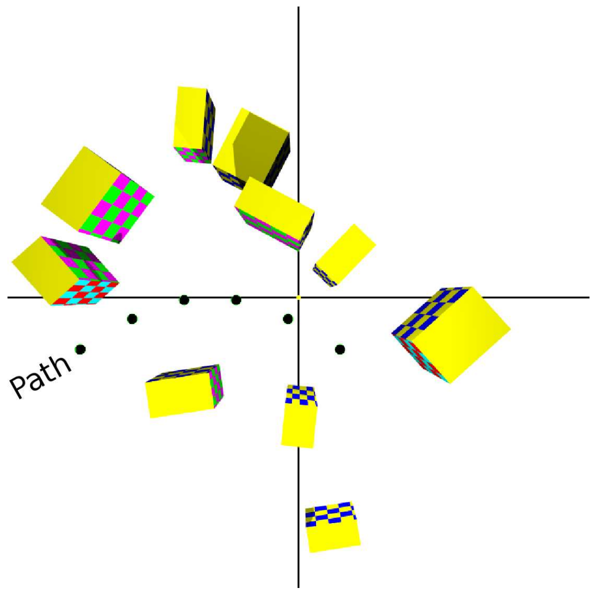


Figure 1.6: Example of a path for an Omnidirectional camera in POV-Ray.

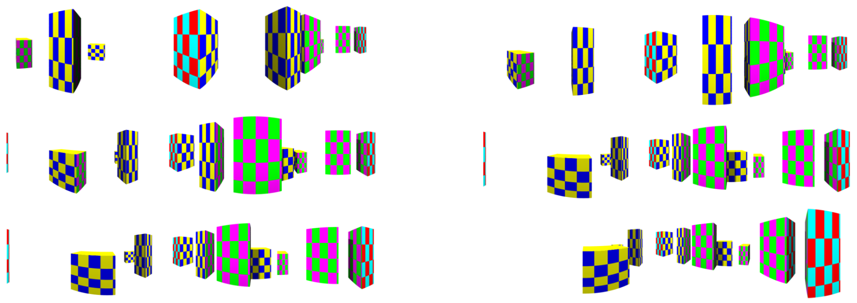


Figure 1.7: Sequence of pictures taken with an Omnidirectional camera in POV-Ray.

The thesis goal

This thesis aims to improve the robustness and efficiency of computing 3D Models using SFM for Omnidirectional cameras. Working in a **semi synthetic environment** allows controlling the error and accuracy of the **structure from motion** reconstruction.

Applications

Possible applications of the result are autonomous **robot navigation**, 3D **motion tracking**, **model reconstruction**, **city reconstruction**, **camera calibration**, **augmented vision**, or **perceptual computer interfaces**.

Advantages of OMNIVIS over other methods of reconstruction

Many methods to build 3D models rely on the use of heavy technologies such as scanners, arrays of cameras with known positions, laser range finders, arrays of sonar sensors and GPS sensors. The structure from motion problem only needs the visual information from a camera to reconstruct both the camera positions and the point locations. It not only renders a 3D model but also find a path in which the camera moved while it recorded the pictures. This is useful in robotics, because it allows a robot with a single camera to produce maps and navigate.

1.1 Prior work on SFM for Omnidirectional vision

The main objective of this section is to give an overview of the prior work on the solution of the structure from motion problem from video taken by a single Omnidirectional catadioptric system combining one **parabolic**, **spherical** or **hyperboloidal** mirror with a perspective camera. The second objective is to mention the prior work on the solution of the structure from motion problem for image sequences taken by alternative **Omnidirectional systems**, such as:

- Omnidirectional stereo systems - a pair of Omnidirectional camera systems.

- Omnidirectional systems consisting of arrays of multiple perspective cameras.
- Hybrid Systems: Omnidirectional systems combined with other sensors.
- Non-traditional cameras.

These latest systems mentioned above, represent expensive and more complex approaches used to solve the **structure from motion** problem. We believe that being aware of this work is important because it can provide us with ideas that we can use later in this thesis.

This section is organized in four parts: the first part explains our previous theoretical work in the **structure from Motion problem**, the second part is to show the comparison of the different approaches using a single catadioptric system to reconstruct exclusively from the pictures, and the third part shows that reconstructions have been done using heavy equipment such as camera pairs, arrays of multiple cameras sometimes combined with information coming from other sensors such as odometers, laser range finders, sonar sensors, light sensors, GPS sensors, networks, etc. And the last part shows non-traditional approaches to the problem.

Part 1: On the Structure from motion problem

In our previous work [Knill and Ramirez-Herran, 2007c, Knill and Ramirez-Herran, 2007a, Knill and Ramirez-Herran, 2007b] using theoretical cameras and infinite precision measurements, our structure from motion results give sharp conditions under which the reconstruction is unique. For example, if there are three points in general position and three Omnidirectional cameras in general position, a unique reconstruction is possible up to a similarity. We then looked at the reconstruction problem with m cameras and n points, where n and m can be large and the over-determined system is solved by least square methods. The reconstruction is robust and generalizes to the case of a dynamic environment where landmarks can move during the movie capture.

Part 2: On single Omnidirectional catadioptric systems.

In an article from [M. Bosse and Teller, 2003] we found how to produce a trajectory map from Omnidirectional video. In [Y. Yagi and Yachida, 2000], an Ego-motion parameter estimation is done for a roaming robot. In the last case they used a Camera with Hyperboloidal Mirror to produce a 2D map of the structure and the robot position.

[Winters et al., 2000] presents a method to transform Omnidirectional images to Bird-Eye-Views which correspond to scaled orthographic views of the ground plane. Navigation is performed by using bird-eye views to track landmarks on the ground plane and estimate the robot's position. The relevant features to track and the feature coordinate system are initialized by the user.

Observing the work summarized in Table 1.1, found at the end of this chapter, we notice that most of the previous work in this case is focused on the task of robot navigation, where you can simplify the problem using lower level features extracted from gray scale pictures and they reconstruct the path mostly. The structure does not need to be seen more like an obstacle or boundary for navigation.

Part 3: On multiple camera systems

In table 1.2, we show the prior work on reconstructions that have been done using pictures from camera pairs or arrays of multiple cameras which sometimes are combined with the use of other sensors such as odometers, laser range finders, sonar sensors, compass sensors, light sensors, GPS sensors, etc. The extra information obtained from other sensors can reduce the number of unknowns in the equations and can be used to make corrections of the calculations.

Etoh et al [M. Etoh and Hata, 1999] use a panoramic camera that consists of six CCD cameras and mirrors to produce a seamless image of 3520 x 576 pixels. They extract a set of vertical lines. Rawlinson and Jarvis [Rawlinson and Jarvis, 2008] use a generalized Voronoi diagram to extract useful topological features. This is especially directed to the task of avoiding obstacles in robot navigation. Note that synthetic environments had been used here. They provide an interactive way of experimenting with the reconstruction in a controlled way. Padja et al [A. Torii, 2009] present a structure from motion pipeline to process Omnidirectional images taken by a camera array called Ladybug to demonstrate a large scale reconstruction using the Google Street View Pittsburgh Research data set.

Part 4: On Object recognition and Hybrid Cameras

In small scale object recognition applications of **structure from motion** methods, a Probabilistic Feature-based On-line Rapid Model Acquisition called PROFORMA propose a pipeline for on-line 3D model reconstruction. As the user rotates the object in front of a stationary camera, a partial model is reconstructed and displayed to the user who assists tracking the pose of the object. Models are rapidly produced through a **Delaunay tetrahedralisation** of points obtained from on-line structure from motion estimation, followed by a probabilistic tetrahedron carving step to obtain a textured surface mesh of the object" [Pan et al., 2009].

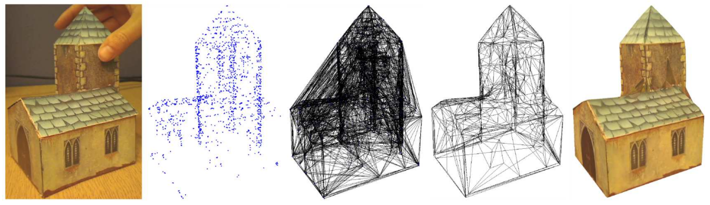


Figure 1.8: Probabilistic Feature-based On-line Rapid Model Acquisition

Another different kind of system that combines 2 cameras: 1 perspective and 1 Omnidirectional [Sturm, 2002] and [PuigLuis et al., 2008] could potentially contribute to the

discussion and it is worthy to explore it later. They are interested in the possibility of factorization-based methods for 3D reconstruction from multiple catadioptric views.

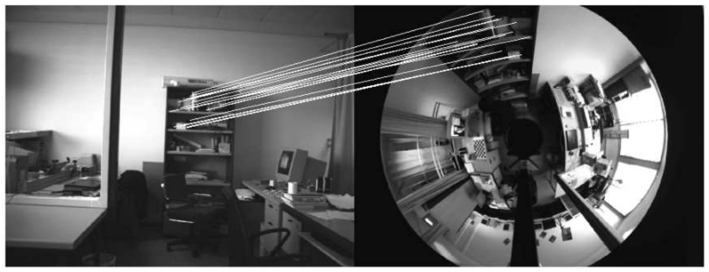


Figure 1.9: Matching SIFT features in Catadioptric and Perspective pictures

Remarks on the previous work

While most research has been done with perspective cameras, also alternative types of cameras have been included. There are advantages of the Omnidirectional camera model. Omnidirectional pictures are becoming popular and they are used to produce unwrapped rectangular panoramas, and new ways of map navigation such as Google Street view. Especially, since Omnidirectional vision sensors have become more affordable, and provide the wider field of view, the structure from motion problem for Omnidirectional cameras have attracted more research. The hope is to figure out simpler, more precise and more affordable methods of reconstructing 3D from video. "Omnidirectional images facilitate landmark based navigation, since landmarks remain visible in all images, as opposed to a small field-of-view standard camera"[Winters et al., 2000].

The tables 1.1 and 1.2 show that our method is less constraining in the number of dimensions reconstructed than the majority of the surveyed methods and it promise to be a more robust and compact solution. Also we will reconstruct assuming that the only information available is from the pictures.

General guidelines

Oliensis [Oliensis, 2000] gave some of the most important guidelines to produce high quality work when one is involved in the complex task of solving the Structure from Motion (SFM) problem. In the same paper he criticized the previous work on SFM for the lack of rigorous analysis and the lack of more meaningful results. Additionally, he explains in detail why the reconstruction algorithms should be tested on a large number of synthetic sequences and he argues that "one can clearly achieve a deeper understanding of the reconstruction sub-problem than of the full problem including correspondence".

To follow this guidelines we are committed to test the algorithms on enough number of sequences, base our experiments and algorithm design on theoretical analysis of algorithm behavior and on an understanding of the intrinsic, algorithm-independent properties of SFM optimal estimation.

1.2 Organization of this document

In chapter 2 we introduce Computer vision and image processing and give details of the implemented algorithms for corner detection. In chapter 3 we introduce the basic concepts on Omnidirectional Cameras. In chapter 4 we explain the mathematics of structure from motion and provide numerical examples. In chapter 5 we explain feature tracking and 3D reconstruction showing examples of the experiments with both synthetic and real scenes. In chapter 6 we give an overview of the software implementation. In chapter 7 we summarize our findings and make conclusions about the work done. At the end of this document, we provide appendices with the source code.

Table 1.1: 3D reconstruction using different catadioptric systems

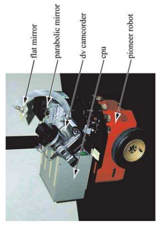
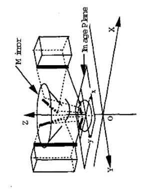
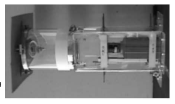
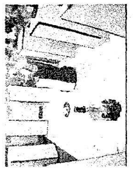
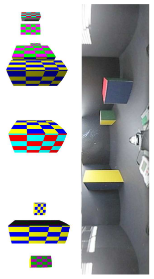
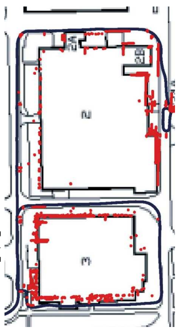
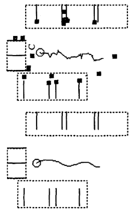
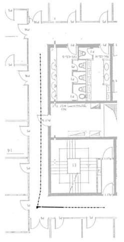
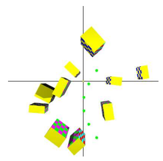
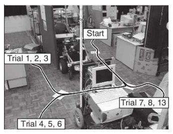
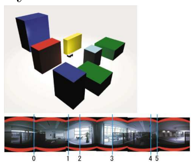
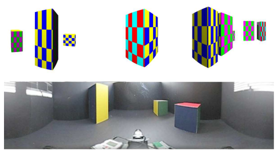
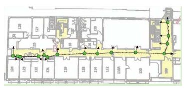
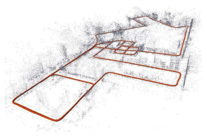
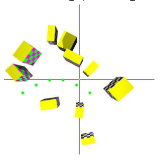
Parameter				
Sensor type	Bosse et al Parabolic	Yagi et al Hyperbolidal	Winters et al Spherical	Ramirez et al Parabolic
Sensor picture	 flat mirror parabolic mirror dv camcorder cpu pioneer robot			
Input type	Omnidirectional	Panorama	Bird's Eye view	Panorama
Scene type	Corridor	Room	Corridor	Synthetic and real
Scene				
Features	Vanishing points, lines	Vertical edges	Edges and corners	Corners
Tracking method	State estimation	Estimating error	Search corners, edges	Search gradient changes
Application	Robot motion, 2D map	Robot motion, 2D map	Robot localization 2D	3D map, 3D path
Structure type	2D map, path	2D map, path	2D path	3D map, 3D path
Structure				

Table 1.2: 3D reconstruction using multiple camera systems

Parameter	Rawlinson et al	Etho et al	Padja et al	Ramirez et al
Sensor type	catadioptric and laser	Six mirrors six cameras	Multicamera array	Parabolic
Sensor picture				
Input type	Panorama	Panorama	panorama	Panorama
Scene type	Corridor and outside	Synthetic and real	Real City	Synthetic and real
Scene				
Features	Voronoi Diagram, SIFT	Verical lines	SURF	Corners
Tracking method	SLAM Manual Landmarks	Manual Landmarks	PROSAC	Searching gradient changes
Application	Robot localization, 2D map	Robot Ego-motion, 2D map	Camera path, 3D map	3D map and camera path
Structure type	2D map and path	2D map, path	2D path	3D map, 3D path
Structure				

Chapter 2

Image Processing and Computer Vision

Computer vision cannot exist without image processing. In this chapter we explain image processing and computer vision methods. The reconstruction that is done in this thesis is based in the correspondences of features extracted using image processing algorithms on color images. While the central problem of **computer vision** is to extract information from image data, in **image processing**, the focus is on transforming images. For instance, extracting a three-dimensional model from two-dimensional images is a **computer vision problem** rather than an **image processing problem**. On the other hand, **image enhancement** is more an image processing problem and less a **computer vision problem**.

2.1 Image processing

Image processing involves processing an image in a desired way or extract interesting features from an existing image to be use in more complex image operations. The first step is obtaining an image in a readable format. The Internet and other sources provide countless images in standard formats. In this thesis we use the JPG, PNM, PNG, and PS file formats discussed in appendix Image Formats.

2.2 Movies

A **movie** is a sequence of images called **frames** with an optional additional sound channel. Each frame is a color picture of fixed width and height. Typical videos have 640 pixel width and 480 pixels height. For our synthetic pictures, we record in 1200 pixel width and 600 pixels height. The **frame rate** is the number of frames per second such as 24 frames per second. Similarly, sound is sampled with a sound sampling rate, usually 44100 times per second. We do not deal with sound in this project and only process the image frames.

2.3 Computer vision methods

Computer vision systems operate on digital images which are quantized, both in space and in intensity. The discrete spatial locations are called **pixels**. They are arranged on a rectangular grid. In a gray-level picture, each pixel takes on a range of integer values called intensities. For a color picture, each pixel is associated a color vector (r, g, b), where r is the read part, g is the green part and b is the blue part. Computer vision

methods are often classified into **low, middle and high levels** according to the table 2.1 [Tucker, 2004]:

Low-level vision techniques are those that operate directly on images and produce outputs that are other images in the same coordinate system as the input.	For example, an **edge detection algorithm** takes an intensity image as input and produces a binary image indicating where edges are present.
Middle-level vision techniques are those that take images or the results of low-level vision algorithms as input and produce outputs that are something other than pixels in the image coordinate system.	For example, a **structure from motion** algorithm takes as input sets of images and produces as output the three dimensional coordinates of those features.
High-level vision techniques are those that take the results of low or middle vision algorithm as input and produce outputs that are abstract data structures.	For example, a model-base recognition system can take a set of image features as input and returns the geometric transformations mapping models in its database to their locations in the image.

Table 2.1: Computer vision methods

2.4 Low-level vision techniques

Low-level features are those basic features that can be extracted automatically from an image without any knowledge about the shape or information about spatial relationships in the picture. A special case is **thresholding**, where a gray scale or color image is transformed into a binary image, where the value is black if some local feature is above a certain threshold value. If the thresholding is done in a good way, to mark interesting points in the picture. These points can then be used for further processing.

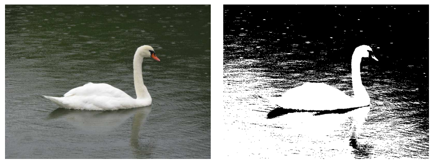


Figure 2.1: Thresholding, an example of image processing. From left to right: Original picture, binary image

In general, Low-level vision computations include tasks such as: finding intensity edges in an image, representing images at multiple smoothing scales, smoothing the image with different filters, compute averaged data such as center of mass for each color, and analyzing the color information in images.

In this section we want to highlight two important subsets of low-level vision techniques. First, we show very basic operators from which the most important is the **Gaussian operator** used for image smoothing. Second, we consider the problems of **edge detection**, and **corner detection** in more detail. The corner and edges are going to be used for the 3D reconstruction (Middle-Level Vision).

2.4.1 Low-level vision techniques: Basic operations

Histograms.

The intensity histogram shows how individual brightness levels are occupied in an image [Nixon and Aguado, 2002]. Contrast is defined as the range of the brightness levels. Histogram is a plot of the number of pixels that each brightness level contains. As example, for 8 bit pixels the brightness is between zero and 255. A histogram can show for example,

whether all available grey levels have been used. This helps to decide whether the image intensities towards have to be modified using more of them and try to see if the image becomes clearer. It this sense, we can say that the histogram can reveal the presence of noise in the image. For later feature extraction techniques it is important not only to improve the appearance but also remove some of the noise.

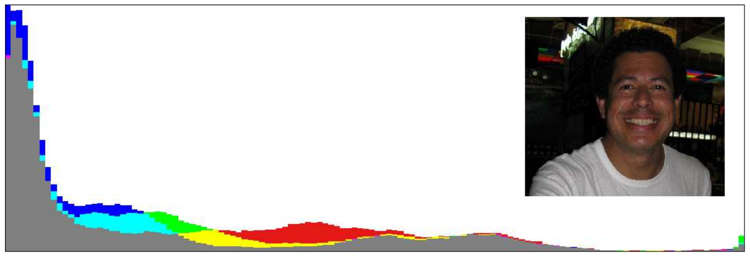


Figure 2.2: Example of histogram. From left to right: Original image, histogram

Point operations.

Each pixel value is replaced with a new value obtained from the old one. Examples of point operations are: Brightness and Thresholding.

Brightness.

If we want to increase the brightness it is sufficient to multiply each pixel value by a scalar. If we want to reduce the contrast, we can divide by a scalar. Basic brightness operations include inversion, addition, logarithmic compression, exponential expansion [Nixon and Aguado, 2002].

Thresholding.

This operator selects pixels that have a particular value or that are within an specified range. For example, given an image of a face we can separate the facial skin from the background; the cheeks, forehead are separated from the hair and eyes.

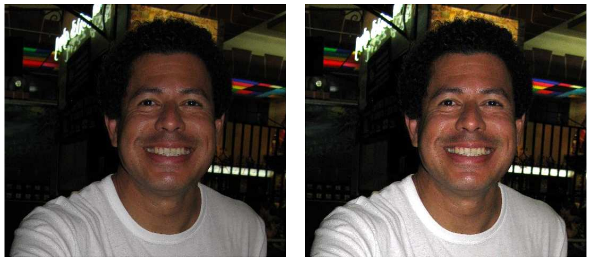


Figure 2.3: Example of Brightness operator - Lighter. From left to right: Original picture, lighter picture

Group operations.

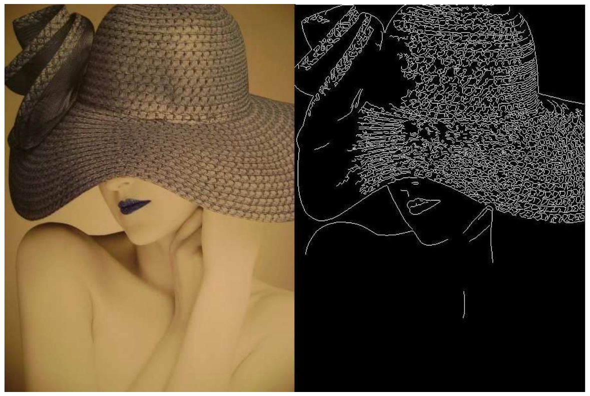


Figure 2.4: From left to right: original, binary image. Courtesy: Candice Rexford.

They calculate new pixel values from a pixel neighborhood. This process is called template convolution. The operation is usually performed by a square template of weighting coefficients. New pixel values are obtained by placing the template at the point of interest and the pixel values are multiplied by the coefficients and added to an overall sum which becomes the new value for the centre pixel. An important operator that uses template convolution is the Gaussian averaging operator.

Gaussian averaging operator

This operator is considered optimal for image smoothing. The values of the weights on

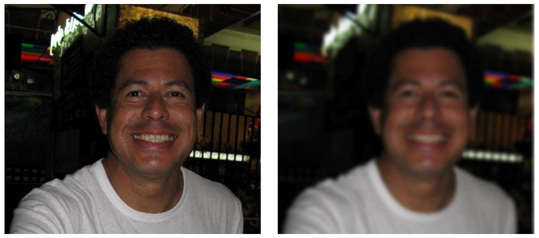


Figure 2.5: Example of Gaussian. From left to right: Original picture, Gaussian $\sigma = 5$

the template are set by the Gaussian relationship. The Gaussian function g at coordinates x, y, is controlled by the variance according to:

$$G(x) = \frac{e^{-\frac{X^2+Y^2}{2\sigma^2}}}{2\pi\sigma^2}$$

2.4.2 Low-level vision: Edge and corner operations

Here we focus is on **corner detection** since corners are distinctive image points which can be located accurately and recur in successive images. This allows us to track them over time. Corners are also often more abundant than straight edges in the real scenes which makes them ideal to track in a real situations.

We use first and second order derivatives of the image intensity to extract low-level features. An example of first order low-level feature is **edge detection**. A local mask is used to determine places where the gray level has large variations. If the variation is above a certain threshold, a black pixel is drawn. This can be used to produce a line drawing which resembles a caricaturist's sketch, although without the exaggeration a caricaturist would use [Nixon and Aguado, 2002]. Such first order detectors only use first derivatives of the gray level density.

An example of a second order low-level feature is **corner detection**. It generally uses second derivatives of the gray level density. This detects points where lines bend very sharply with high curvature. In the next section we expand the subject of edge detection.

2.5 Local edge detectors

The goal is to extract geometric information contained in an image. Many physical events can cause image intensity changes or **edges** in an image. Only some of these are geometric like object boundaries and surface boundaries. Other intensity changes indirectly reflect geometry changes like **specular reflections**, **shadows** and **inter-reflections**.

We will refer to a gray level image as $I(x, y)$, which denotes intensity as a function of the image coordinate system. Intensity edges correspond to rapid changes in the value of $I(x, y)$. Everything described here for $I(x, y)$ can also be done for the red, green and blue channels of the picture. We actually do everything in color space for our reconstruction. But since the method is the same in each color coordinate, we can discuss a single function $I(x, y)$ instead.

To detect this changes is common to use differential properties such as the **squared gradient magnitude**

$$\|\nabla I\|^2 = (\frac{\partial I}{\partial x})^2 + (\frac{\partial I}{\partial y})^2 \ .$$

A large squared gradient magnitude suggests the presence of an edge. Another local differential operator is the **Laplacian**

$$\Delta I = |\nabla|^2 I = (\frac{\partial^2 I}{\partial x^2}) + (\frac{\partial^2 I}{\partial y^2})$$

This second derivative operator preserves information about which side of an edge is brighter. The **zero crossings** are the sign changes of $\nabla^2 I$ correspond to intensity changes in the image, and the sign on each side of a zero crossing indicates which side is brighter.

The images used in computer vision are digitalized in both space and intensity, producing an array $I(j, k)$ of discrete intensity values. Thus, in order to compute local differential operators, **finite difference approximations** are used to estimate the derivatives.

One of the best discretizations is called **Sobel** partial derivatives. We use them often in our work. The **Sobel partial derivative** is defined as

$$\begin{aligned}
\partial_x f(x,y) &= \frac{2}{4}\,[f(x+1,y) - f(x-1,y)] \\
&+ \frac{1}{4}\,[f(x+1,y+1) - f(x-1,y+1)] \\
&+ \frac{1}{4}\,[f(x+1,y-1) - f(x-1,y-1)] \ .
\end{aligned}$$

It is just a weighted sum of symmetric partial derivatives, with the net effect that an additional average takes place in the y direction. Similarly, we have the Sobel partial y-derivative:

$$\begin{aligned}
\partial_y f(x,y) &= \frac{2}{4}\,[f(x,y+1) - f(x,y-1)] \\
&+ \frac{1}{4}\,[f(x+1,y+1) - f(x+1,y-1)] \\
&+ \frac{1}{4}\,[f(x-1,y+1) - f(x-1,y-1)] \ .
\end{aligned}$$

2.6 Local corner detectors

There are various corner detectors known. We studied Kitchen-Rosenfeld, Harris, SUSAN, and KLT and used Kitchen-Rosenfeld for our work.

2.6.1 Cross corner detector

Lets start with a primitive toy corner detector, which detects horizontal or vertical corners. If you look at most pictures, interesting points are often points, where vertical and horizontal lines cross. Since in many real life applications such as city reconstruction, most edges are horizontal or vertical, we implemented this kind of detector in our code. These are points, where $|f_{xy}|$ is large. In the following figure, we show an example of detecting horizontal crosses: places where the x and y derivatives are both large. In real scenes like in the figure, is typical to see many horizontal or vertical corners. We draw them by hand for illustration purposes. are .

2.6.2 Kitchen-Rosenfeld Corner Detection

This method looks at the level curves of $I(x, y)$ and computes the curvature of those curves. If the curvature is large and additionally the edge detector also signals a large gradient, then we have a corner. This method can be explained using smooth functions $I(x, y)$. In the discrete case, we just have to replace the usual partial derivatives with Sobel discretizations.

The curvature of I is the rate of change of the direction of the gradient vector in the direction parallel to the level curve. In other words, as taught in multivariable calculus,

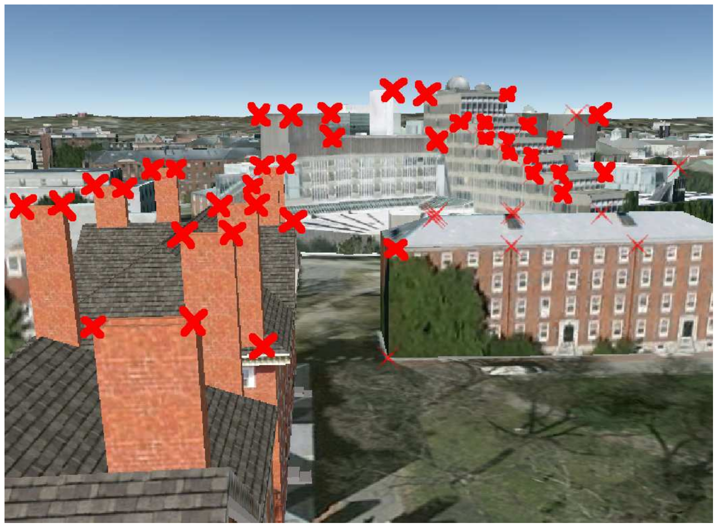


Figure 2.6: Example of cross corners in a typical real picture. Google Earth

it is $D_v\alpha$, where v is a vector parallel to the level curve and α is the angle of the gradient. In the following figure, we show the gradient vector to a level curve of $I(x, y)$ and the direction tangent to the level curve in the first picture. In the second picture, we show the gradient computed at points where the Kitchen-Rosenfeld curvature is large in a synthetic scene. In the third picture, we calculate the gradient in a real picture: The Memorial hall at Harvard University.

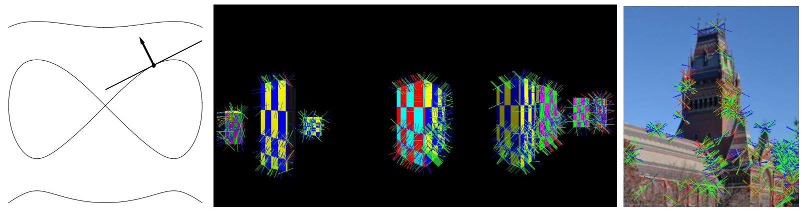


Figure 2.7: The gradient and Examples of using Kitchen-Rosenfeld Corner Detection

The angle of the gradient vector is

$$\alpha(x,y) = \arctan(\frac{f_y}{f_x}) ,$$

where we use the short hand notation $f_x = \partial f(x,y)/\partial x$ and $f_y = \partial f(x,y)/\partial y$.

Let $v = \langle -f_y, f_x \rangle$ be a vector normal to the gradient. The curvature is

$$K(x,y) = D_v\alpha(x,y) = v \cdot \nabla\alpha = \frac{1}{\sqrt{f_x^2 + f_y^2}} \langle -f_y, f_x \rangle \cdot \langle \alpha_x, \alpha_y \rangle .$$

We have now to compute

$$\alpha_x = \frac{\partial\alpha(x,y)}{\partial x} = \frac{\partial}{\partial x} \arctan(f_y/f_x)$$

and

$$\alpha_y = \frac{\partial\alpha(x,y)}{\partial y} = \frac{\partial}{\partial y} \arctan(f_y/f_x) .$$

Simplification gives

$$\alpha_x = \frac{f_x(x,y)f_{xy}(x,y) - f_y(x,y)f_{yy}(x,y)}{f_y(x,y)^2 + f_x(x,y)^2}$$

$$\alpha_y = \frac{f_y(x,y)f_{xy}(x,y) - f_x(x,y)f_{xx}(x,y)}{f_y(x,y)^2 + f_x(x,y)^2}$$

so that

$$\langle -f_y, f_x \rangle \cdot \langle \alpha_x, \alpha_y \rangle = \frac{f_{yy}(x,y)f_y(x,y)^2 - 2f_x(x,y)f_{xy}(x,y)f_y(x,y) + f_{yy}(x,y)f_x(x,y)^2}{f_x(x,y)^2 + f_y(x,y)^2}$$

and

$$K(x,y) = \frac{f_{yy}(x,y)f_y(x,y)^2 - 2f_x(x,y)f_{xy}(x,y)f_y(x,y) + f_{yy}(x,y)f_x(x,y)^2}{(f_x(x,y)^2 + f_y(x,y)^2)^{(3/2)}} .$$

We have proven:

Theorem 2.1 *The curvature of a level curve $I(x,y) = c$ at a point is given by the formula*

$$K(x,y) = \frac{f_{xx}f_y^2 - 2f_{xy}f_xf_y + f_{yy}f_x^2}{(f_x^2 + f_y^2)^{3/2}} .$$

The following three figures show examples of the application of our implementation of the Kitchen-Rosenfeld Corner Detection to showing the gradient in the three color channels in different kind of scenes.

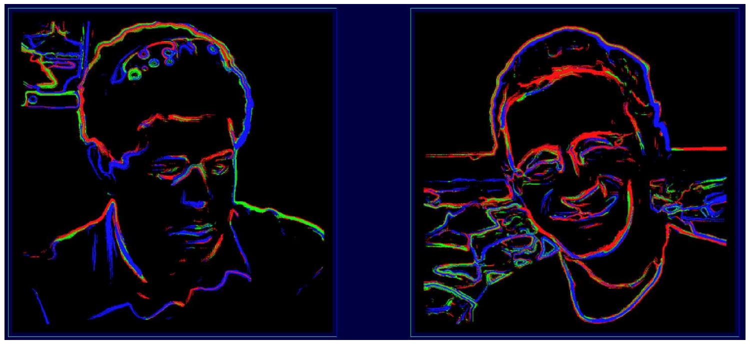


Figure 2.8: Gradient applied to two different pictures with the same smoothing levels.

The information of the calculations of the curvature and gradient using the Kitchen-Rosenfeld Corner Detection can be stored in a separate pictures whihc can be used later to speed up the program or simply for illustration.

Here is a short overview over other corner detectors:

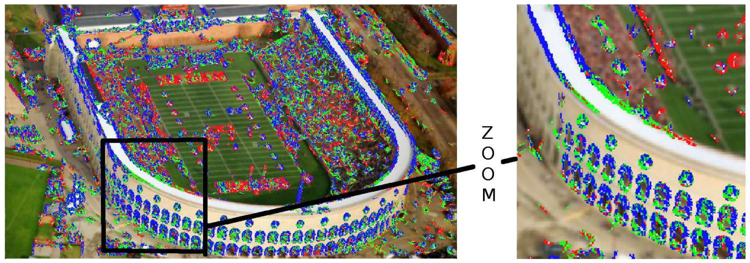


Figure 2.9: Computing vectors tangent to the level curves

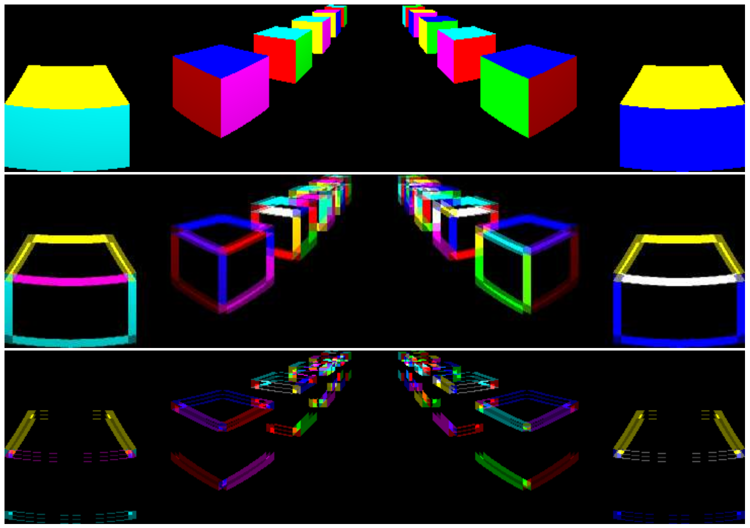


Figure 2.10: Curvature and gradients in a synthetic scene.

2.6.3 Harris Corner Detection

The Harris corner detector is an algorithm based on an underlying assumption that corners are associated with maxima of the local autocorrelation function. It is less sensitive to noise in the image than most of the other algorithms because the computations uses first derivatives of $f(x, y)$ only. Image smoothing is required to improve the performance of the detection leading however to poor localization accuracy. The Harris corner detector

introduces a **cornerness** value

$$c = \frac{< f_x^2 > + < f_y^2 >}{< f_x^2 >< f_y^2 > - < f_x f_y >^2}$$

for each pixel where

$$< f > (x, y) = \frac{1}{(2r+1)^2} \sum_{|i|<r, |j|<r} f(x+i, y+j)$$

denotes a **spacial average** of the function f over some square neighborhood of size $(2r + 1) \times (2r + 1)$. A pixel is declared a **Harris corner** if the value c is below a certain threshold. The size of the neighborhood, (the radius r) required to calculate c is determined by the size of the Gaussian smoothing kernel. For a $(2n+1) \times (2n+1)$ kernel, one takes a $(2n + 3) \times (2n + 3)$ neighborhood. For example, for a 3×3 kernel (where $n = 1$), one takes a 5×5 neighborhood (where $r = 2$) [P. Tissainayagam, 2004].

The **Harris detector** uses the same idea as a more general **Moravec detector** [Nixon and Aguado, 2002]. The idea is to look how the average image intensity changes when a window is shifted in several directions. This is measured by a quantity called **autocorrelation of** f **in the direction** (u, v) which is defined as

$$E_{u,v}(x, y) =< (f(x, y) - f(x + u, y + v))^2 > \ .$$

A measure of curvature is the minimal value of $E_{u,v}$ where (u, v) is one of the 4 main directions. The Harris detector replaces $f(x + u, y + v) - f(x, y)$ by the directional derivative $D_{(u,v)} f = f_x u + f_y v$ leading to

$$E_{u,v}(x, y) =< f_x u + f_y v >^2 \ .$$

The maximal change is when (u, v) is proportional to the gradient $< f_x, f_y >$ because moving in the direction of the gradient produces maximal ascent. This leads to the autocorrelation

$$E(x, y) =< f_x^2 > + < f_y^2 > \ .$$

[Nixon and Aguado, 2002] (page 161) give argument why this is divided by $< f_x^2 >< f_y^2 > - < f_x f_y >^2$ to get c.

Another approach is to consider f_x and f_y as **random variables** and the gradient (f_x, f_y) a random vector if the averages $E[f_x] =< f_x >, E[f_y] =< f_y >$ are zero, then $\mathrm{Var}[f_x] =< f_x^2 >$ is the **variance** of f_x and $\mathrm{Var}[f_x] =< f_y^2 >$ is the variance of f_y and the **covariance** between f_x and f_y is

$$\mathrm{Cov}[f_x, f_y] = E[(f_x - < f_x >)(f_y - f_y)] = E[f_x f_y] =< f_x f_y > ,$$

then the **covariance matrix** of the random gradient vector is

$$C = \begin{bmatrix} < f_x^2 > & < f_x f_y > \\ < f_x f_y > & < f_y^2 > \end{bmatrix} .$$

This covariance matrix C captures the intensity structure of the local neighborhood [Derpanis, 2004]: if the eigenvalues of C are both small, the auto-correlation function is flat. If one eigenvalue of C is large and the other small, then local shifts in one direction cause little change while a significant change happens in the orthogonal direction. If both eigenvalues are large, the auto-correlation function is sharply peaked. Shifts in any direction will result in a significant increase. This indicates a corner. Note that the determinant of C is $< f_x^2 >< f_y^2 > - < f_x f_y >^2$ and the trace of C is $< f_x^2 > + < f_y^2 >$ so that the Harris cornerness is equal to $\mathrm{tr}(C)/\det(C)$ which is $(\lambda + \mu)/(\lambda\mu)$ if λ, μ are

the eigenvalues of C. From the eigenvalue equation

$$\lambda_\pm = \frac{\mathrm{tr}(A)}{2} \pm \sqrt{\frac{\mathrm{tr}(A)^2}{4} - \det(A)}$$

we get

$$\frac{\lambda_\pm}{(\lambda_+ + \lambda_-)} = \frac{1}{2} \pm \sqrt{1 - 4\frac{\det(A)}{\mathrm{tr}(A)}} = \frac{1}{2} \pm \sqrt{1 - \frac{4}{c}} \ .$$

So

$$\frac{8}{c} = \frac{1}{2} + \frac{\lambda_+ - \lambda_-}{\lambda_+ + \lambda_-}$$

which shows that if $\lambda_+ - \lambda_-$ is large, then c is small.

2.6.4 Smith Univalue Segment Assimilation Nucleus

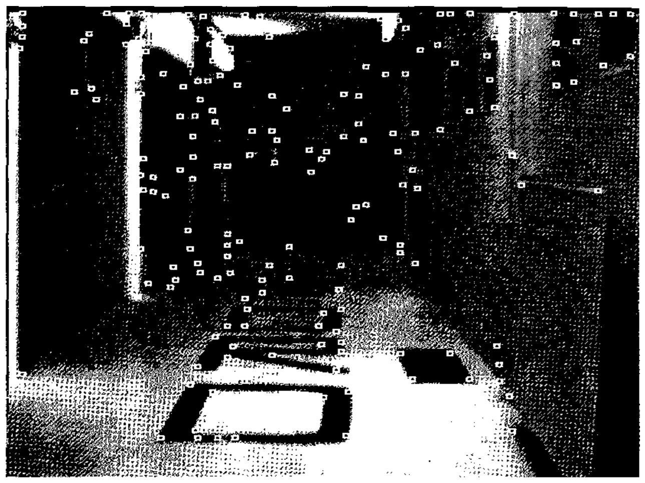


Figure 2.11: Extracting interesting points using SUSAN

[Smith and Brady, 1997] developed a very simple corner detector that does not uses spatial derivatives and does not require smoothing. Each pixel in the image is used as the center of a small circular mask. The grayscale values of all the pixels within this circular mask are compared with that of the center pixel (*thenucleus*). All pixels with similar brightness to that of the nucleus are assumed to be part of the same structure of the image and then are colored black, and pixels with different brightness are colored white. Smithe calls the black area the Univalue Segment Assimilating Nucleus (USAN). He argues that the USAN corresponding to a corner has an USAN area of less than a half the total mask area. A local minimum in USAN area will find the exact point of the corner. In practice, the circular mask is approximated using a 5 x 5 pixel square with 3 pixels added on to the center of each edge. The intensity of the nucleus is then compared with the intensity of every USAN area. other pixel within the mask using a comparison function. This comparison is done for each pixel in the circular mask, and a running total, n, of the outputs, c, is made: The total n is 100 times the USAN's area. The USAN area n is the thresholded to extract the corners. A pixel is declared corner if its USAN area, n, is less than half the maximum possible USAN area.

2.6.5 Algebraic corner detection

A recent corner detection has been found by Andrew Willis and Yunfeng Sui. It is called **algebraic model for fast Corner Detection**.

In contrary to the Harris detector, the algorithm considers spacial coherence of the edge points. It uses the fact that the edge points must lie close to one of the two intersecting lines.

The algorithm works in three steps: 1) First the edge image is computed: this is achieved by thinning a blurred version of the picture.

2) Then the function is fitted with hyperbolic conic sections of the form

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

which are hyperbolic, that is for which the discriminant $D = 4ac - b^2$ is -1. This data fitting problem leads to a Lagrange problem.

3) Willis and Sui had to control numerical instabilities by adding extra random noise in some cases.

2.6.6 An application of Corner detection

We chose to use Kitchen-Rosenfeld because it performed better and it was more robust making the tracking easier.

Corner detection can be useful also for Contour tracking in pictures. We find places, where the gradient and curvature is large and then follow the vector field perpendicular to the gradient. Mathematically, the vector field perpendicular to the gradient field is called a Hamiltonian field

$$F(x,y) = \langle -f_y, f_x \rangle = J\nabla f(x,y) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} f_x \\ f_y \end{bmatrix}$$

where $f(x,y)$ is one of the red, green or blue densities. Here is an illustration:

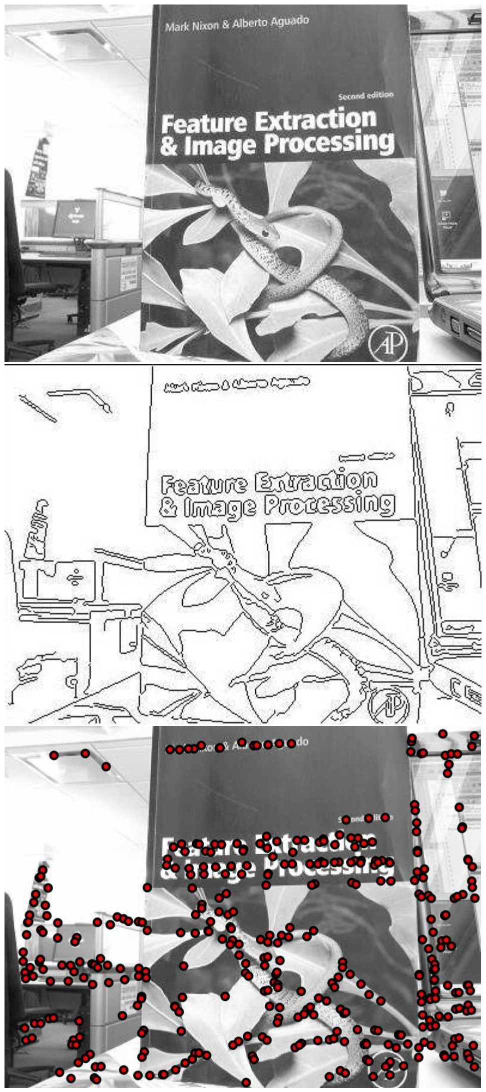


Figure 2.12: Algebraic corner detection steps: Gray levels, "Lines", intersection of "lines". Matlab code provided by Willis and Sui

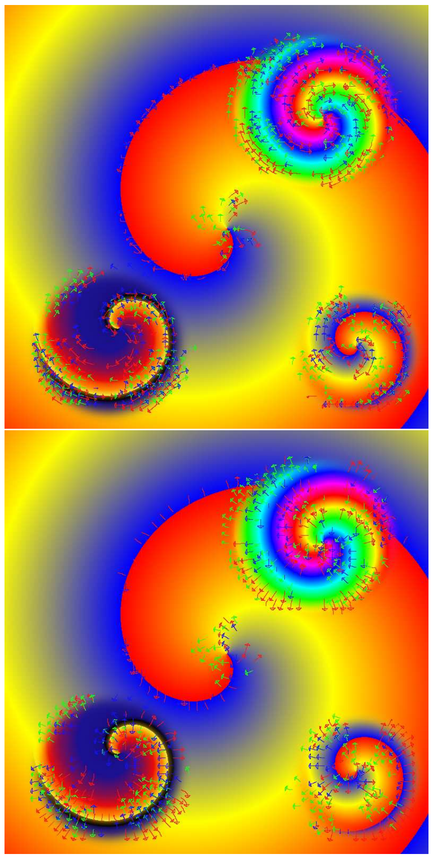


Figure 2.13: Plotting the gradient and Hamiltonian field for each color using our code.

2.7 Middle-level Vision: Structure from Motion

From table 2.1 recall that Middle-level vision techniques take images or the results of low-level vision algorithms as input and produce outputs that are something other than pixels in the image coordinate system. In the case of this thesis we require to produce outputs which are the three-dimensional coordinates of both points and cameras from the corners extracted and their correspondences across frames.

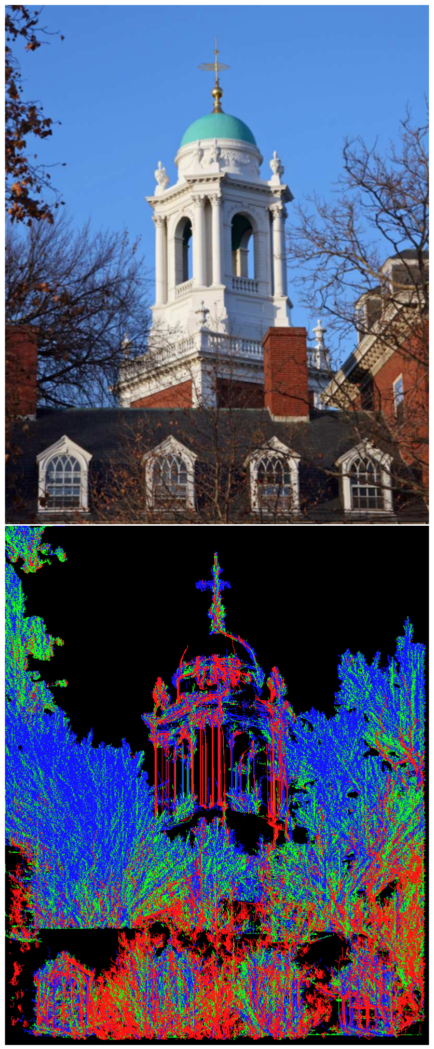


Figure 2.14: Following the Hamiltonian vector field using our code.

Chapter 3

Omnidirectional cameras

We divided this chapter in three sections to explain the following Omnidirectional cameras: The kind of cameras that man has created to provide us with Omnidirectional images, the biological equivalent of an Omnidirectional camera, and the POV-Ray camera used for our experiments with synthetic scenes.

3.1 Man made panoramic vision

Here are some different types of Omnidirectional Vision Sensors (ODVSs) which create Omnidirectional Vision Images (ODVI).

Spherical Omnidirectional mirrors are suitable for observing objects which are at the same height or below the sensor. They come in different types:

- The **conical mirror** is useful to acquire images for which the vertical visual field is limited.
- The **hyperboloidal mirror** in combination with a normal lens can generate images

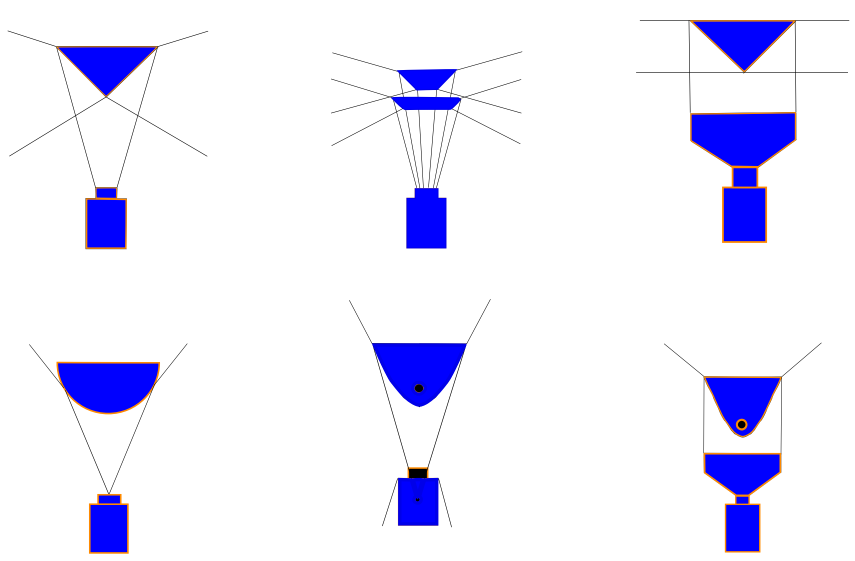


Figure 3.1: Typical mirrors used for Omnidirectional Vision. Illustration from [Benosman and S.B.Kang, 2001].

which can be transformed to normal perspective images. Therefore, it is used for monitoring.

- The **parabola mirror** it is an ideal optical system to acquire Omnidirectional images because the astigmatism is small for a small curvature.

The most interesting systems are the one with a single central projection. The are two of these which result in an imaging configuration with a unique point of projection. They are called ”central panoramic catadioptric cameras”.

- The telecentric lens and paraboloid mirror combination
- The perspective lens and hyperboloid mirror combination.

The main advantages of these systems are:

- They have both horizontal and vertical larger fields of view
- Any distortions can be digitally corrected making them suitable for the task of reconstruction.

The Omnidirectional Vision Sensor used for this project is the most affordable central panoramic catadioptric system in the market and it is called 0-360 One-Click panoramic optic. The camera system is shown in the following figure. This kind of camera allows us to cover a wide field of view with just a single image capture. [Benosman and S.B.Kang, 2001] describes in detail this kind of Omnidirectional camera. This system is a specially designed

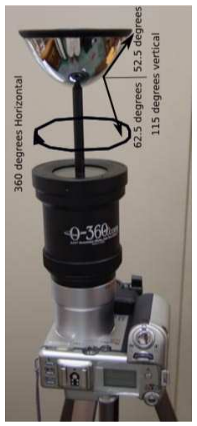


Figure 3.2: 0-360.com panoramic optic

camera panorama lens attachment, with an exclusive optical reflector which captures an entire 360 degree panoramic virtual tour with a single shot. The field of view is 115 degree vertical and 360 degrees horizontal.

3.2 Omnidirectional Vision in Nature

A camera model based on inside-out versions of the reflecting eye of the deep sea crustacean Gigantocypris, as described in [Benosman and S.B.Kang, 2001], inspired in the invention of the catadioptric camera. The term ”catadioptric”, also known as a mirror lens, refers to the use of glass elements (lenses) and mirrors in an imaging system.

Figure 3.3: Panoramic camera by insects. Photo by Giovanni Ramirez

A good overview over Omnidirection vision in nature is [Benosman and S.B.Kang, 2001]: Omnidirection vision exists especially for diurnal and nocturnal insects and some species of crustaceans such as lobsters, shrimps and crawfish.

The deep sea crutacean **Gigantocypris** uses a reflecting mirror which allows the creature to see under very dim conditions.

3.3 Unwrapping an Omnidirectional image

Unwrapping is the process to get from the circular panorama picture taken by the camera to a rectangular picture. This uses polar coordinates.

Figure 3.4: The eyes of the Gigantocypris. Picture credit: Jose Xavier

Coordinate transformation

The picture taken by the camera has the Cartesian coordinates (x, y). If we write this in polar coordinates, we get the point (θ, r) in the unwrapped picture.

The angles are defined by $\theta_{ij} = \frac{y}{x}$

The coordinates x_i and y_i are obtained by

$(x_i, y_i) = r_{ij}\cos(\theta_{ij}), r_{ij}\sin(\theta_{ij})$

Where, the distances r_{ij} are given by

$r_{ij} = \sqrt{(x_i)^2 + (y_i)^2}$

Large r corresponds to points high up in the picture, small r points near the ”foot” of the camera.

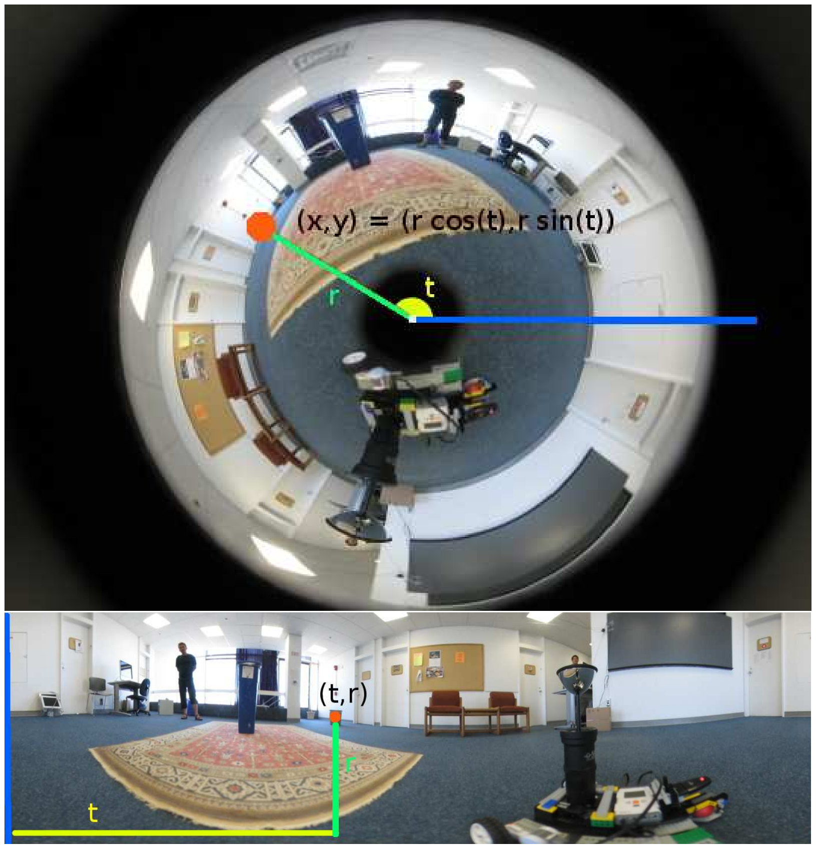


Figure 3.5: Before unwrapping and after unwrapping

3.4 POV-Ray's Omnidirectional Camera

POV-Ray, the persistence of vision ray tracer, is a program that creates three-dimensional, photo-realistic images using a rendering technique called ray-tracing. It reads in a text file containing information describing the objects and lighting in a scene and generates an image of that scene from the view point of a camera also described in the text file. Although this rendering technique is an expensive computational approach, it is easy to

implement and can produce pictures that show proper shadows, mirror-like reflections, and the passage of light through transparent objects [Kelley and Jr, 2007].

3.4.1 Coordinate system

The usual coordinate system for POV-Ray has the positive y-axis pointing up, the positive x-axis pointing to the right, and the positive z-axis pointing into the screen as follows:

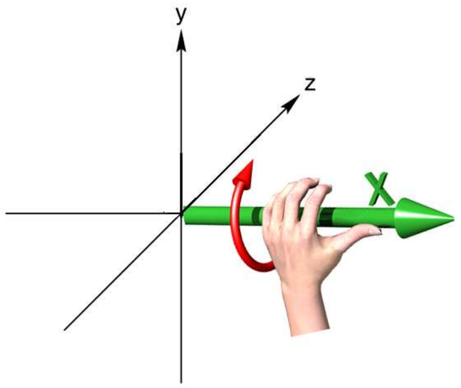


Figure 3.6: Left handed coordinate system. Image source: POV-Ray.org

This is a left-handed coordinate system. The left hand can also be used to determine rotation directions. To do this we hold up our left hand and point our thumb in the positive direction of the axis of rotation. Our fingers will curl in the positive direction of rotation. Similarly if we point our thumb in the negative direction of the axis our fingers will curl in the negative direction of rotation.

3.4.2 POV-Ray Camera

We use the POV-Ray's Omnidirectional camera model to film the scene. POV-Ray allows us to orient the camera using vectors as shown in the following figure.

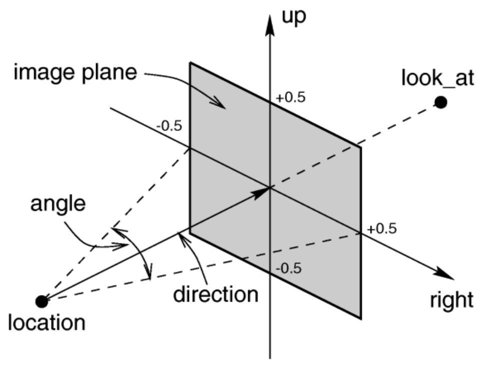


Figure 3.7: Camera parameters in POV-Ray. Image source: POV-Ray.org

Under many circumstances just two vectors in the camera statement are all we need to position the camera: location and look at vectors. For example:

```
camera {
    perspective location <0,0,0>
    look_at <0,2,1>
}
```

The location is simply the x, y, z coordinates of the camera. The camera can be located anywhere in the ray-tracing universe. The look at vector tells POV-Ray to pan and tilt the camera until it is looking at the specified x, y, z coordinates. By default the camera looks at a point one unit in the z-direction from the location.

We use the following statements in this thesis to specify the POV-Ray camera:

```
#declare A=<550,160,200>+<-400*clock,0,0>;
camera{
 panoramic
 up y right x
 location A
 look at A-<0,0,200>
 }
```

Up and Right Vectors

The primary purpose of the up and right vectors is to tell POV-Ray the relative height and width of the view screen. The default values are: right $\frac{4}{3}$ x, up y. In the default perspective camera, these two vectors also define the initial plane of the view screen before moving it with the look at or rotate vectors. The length of the right vector, together with the direction vector, may also be used to control the horizontal field of view with some types of projection. The look at modifier changes both the up and right vectors. The angle calculation depends on the right vector. Most camera types treat the up and right vectors the same as the perspective type. Note that the up, right, and direction vectors should always remain perpendicular to each other or the image will be distorted.

3.4.3 Camera Calibration in POV-Ray

We tested the camera built in POV-Ray by placing spheres on a sphere. The POV-Ray camera is outside the sphere mentioned in the first picture. In the second picture, the camera is in the center of the sphere. The unwrapped picture of the spheres on a sphere shows the regular pattern of the calibrated camera.

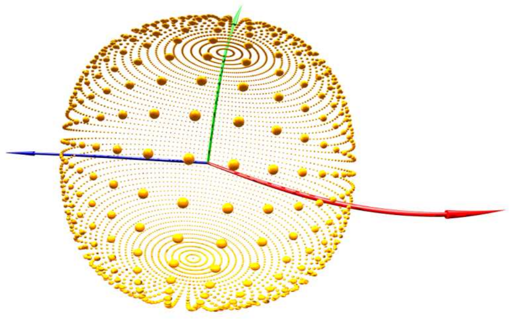


Figure 3.8: Picture taken with an Omnidirectional camera from outside the sphere.

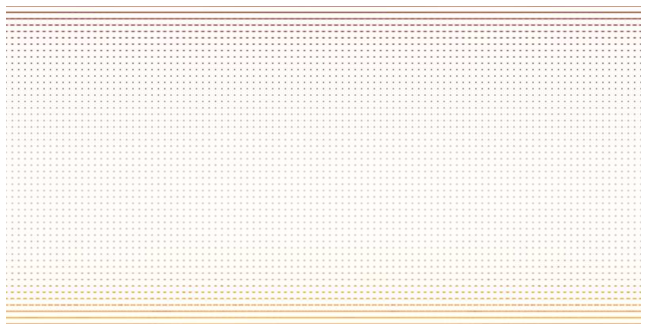


Figure 3.9: This is the picture thee camera sees from the center of the cloud.

In the next chapter we describe the mathematics of the structure from motion and the strategy for the 3D reconstruction.

Chapter 4

Mathematical Model of the 3D Reconstruction

In this chapter we illustrate the mathematics of the reconstruction both in 2D and 3D scenes. We use least-square methods to solve the structure from motion problem. We show explicitly, how to get the equations in matrix form and how can solve the systems of linear equations to recover the information about the scene and the camera positions from the pictures.

Oliensis [Oliensis, 2000] suggests: "real-image experiments are not always the best means of evaluating reconstruction algorithms. In this thesis, we follow this advise and film synthetic scenes and use the pictures to reconstruct the environment and path. He suggests to use a database of tracked correspondences to compare reconstruction algorithms as we did that in previous work [Knill and Ramirez-Herran, 2007b]. Using synthetic scenes with known matched points does not involve the correspondence problem. In this chapter we assume that the correspondences are known, but in this thesis we reconstruct without a priory knowing the correspondences. Still the original scene is synthetic.

In the next chapter we explain how to get this correspondences automatically using what we call tracking.

4.1 Structure from motion

Structure from motion methods allow to recover a three-dimensional model of an object from a sequence of pictures. It can be used in many applications such as robot grasping, robot navigation, medical imaging, and graphical modeling.

Unlike for **stereo vision**, where we know the relative position of two cameras is known, the structure from motion problem reconstructs both the scene and the camera position from several pictures.

What is the difference between structure from motion and SLAM? The Simultaneous Location and Map Building Problem (SLAM) is a version of structure from motion, where the position of the camera is computed in real time while building a map of the environment. The mathematics are different. For structure of motion, we take pictures along a path and then reconstruct the path. In SLAM, the position of the camera is known at any time during the navigation.

4.2 Mathematics of the reconstruction in the planar case

Given n points $P_i = (x_i, y_i)$ and a camera path $r(t) = (a(t), b(t))$. The camera at m position $Q_j = (a_j, b_j) = r(t_j)$ observes the angles $\theta_i(t_j)$.

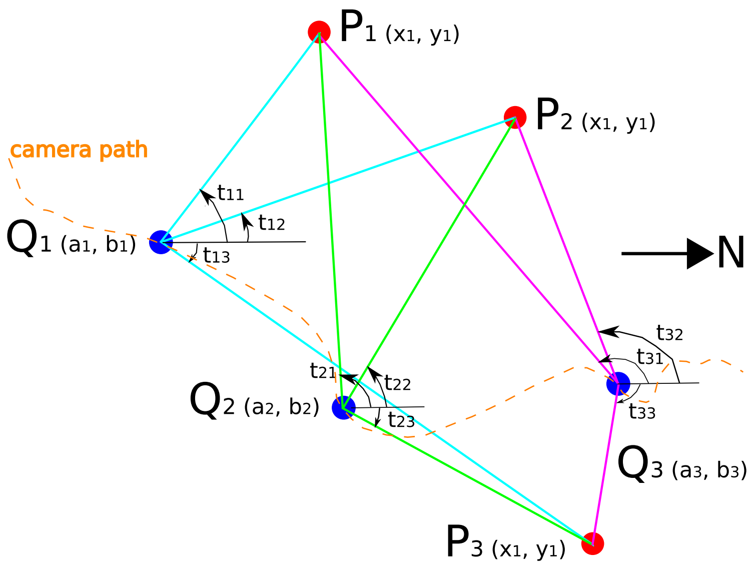


Figure 4.1: General situation of Cameras and Points in the plane

We get a system of nm linear equations

$$\sin(\theta_{ij})(b_i - y_j) = \cos(\theta_{ij})(a_i - x_j)$$

for the $2n$ variables a_j, b_j and $2m$ variables x_i, y_i.

For example, for 3 points and 3 time observations, we can already reconstruct both the 3 points and the 3 camera positions in the plane in general. For $n = 3$ and $m = 3$ there are 2n + 2m 3 = 9 unknowns and mn = 3 3 = 9 equations as we discussed in our previous work [Knill and Ramirez-Herran, 2007b].

The problem is reduced to solve a linear system of equations with n unknowns and n equations, which have an unique solution:

$$X = (A)^{-1} b\ .$$

As example consider the following specific situation in 2D with 3 cameras and 3 points.

Point positions:

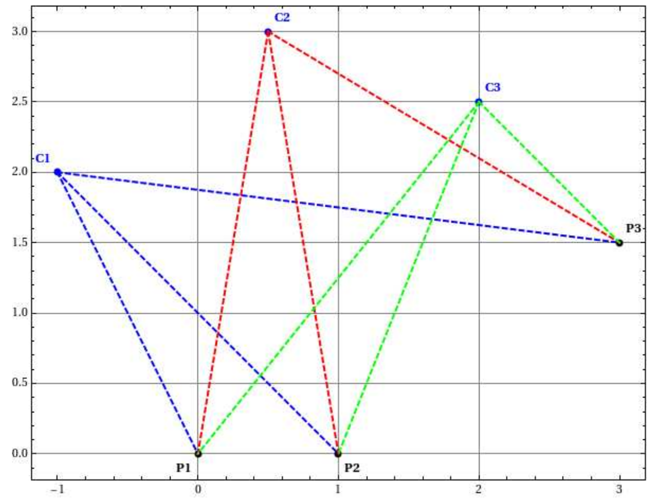


Figure 4.2: Cameras and Points in the plane

$$P_1 = (0, 0)$$

$$P_2 = (1, 0)$$

$$P_3 = (3, \frac{3}{2})$$

Camera positions:

$$C_1 = (-1, 2)$$

$$C_2 = (\frac{1}{2}, 3)$$

$$C_3 = (2, \frac{5}{2})$$

The matrix representation of the system of equations is

$$A = \begin{pmatrix}
-\frac{2}{\sqrt{5}} & 0 & 0 & -\frac{1}{\sqrt{5}} & 0 & 0 & \frac{2}{\sqrt{5}} & 0 & 0 & \frac{1}{\sqrt{5}} & 0 & 0 \\
0 & -\frac{6}{\sqrt{37}} & 0 & 0 & \frac{1}{\sqrt{37}} & 0 & \frac{6}{\sqrt{37}} & 0 & 0 & -\frac{1}{\sqrt{37}} & 0 & 0 \\
0 & 0 & -\frac{5}{\sqrt{41}} & 0 & 0 & \frac{4}{\sqrt{41}} & \frac{5}{\sqrt{41}} & 0 & 0 & -\frac{4}{\sqrt{41}} & 0 & 0 \\
-\frac{1}{\sqrt{2}} & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} & 0 \\
0 & -\frac{6}{\sqrt{37}} & 0 & 0 & -\frac{1}{\sqrt{37}} & 0 & 0 & \frac{6}{\sqrt{37}} & 0 & 0 & \frac{1}{\sqrt{37}} & 0 \\
0 & 0 & -\frac{5}{\sqrt{29}} & 0 & 0 & \frac{2}{\sqrt{29}} & 0 & \frac{5}{\sqrt{29}} & 0 & 0 & -\frac{2}{\sqrt{29}} & 0 \\
-\frac{1}{\sqrt{65}} & 0 & 0 & -\frac{8}{\sqrt{65}} & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{65}} & 0 & 0 & \frac{8}{\sqrt{65}} \\
0 & -\frac{3}{\sqrt{34}} & 0 & 0 & -\frac{5}{\sqrt{34}} & 0 & 0 & 0 & \frac{3}{\sqrt{34}} & 0 & 0 & \frac{5}{\sqrt{34}} \\
0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}$$

And here are the Variables:

$$x = \{a(1), a(2), a(3), b(1), b(2), b(3), x(1), x(2), x(3), y(1), y(2), y(3)\}$$

$$b = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0\}$$

Observe that because scaling and translating of a solution does not change the angles, we can fix $x_2 = 1$ and the distance P_1 and P_2.

So, if $mn \geq 2n + 2m - 3$, we expect a unique solution. If we write the system of linear equations as $AX = b$, in the general case the **least square solution** is

$$X = (A^T A)^{-1} A^T b\ .$$

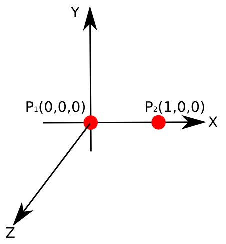


Figure 4.3: Fixing the origin of the coordinate system and fixing a scale

This is the solution of the structure of motion problem. The vector X contains both the point coordinates as well as the camera positions.

In the following example we demonstrate this overdetermined case, for which is found a unique solution by using the scaling and translating constraints.

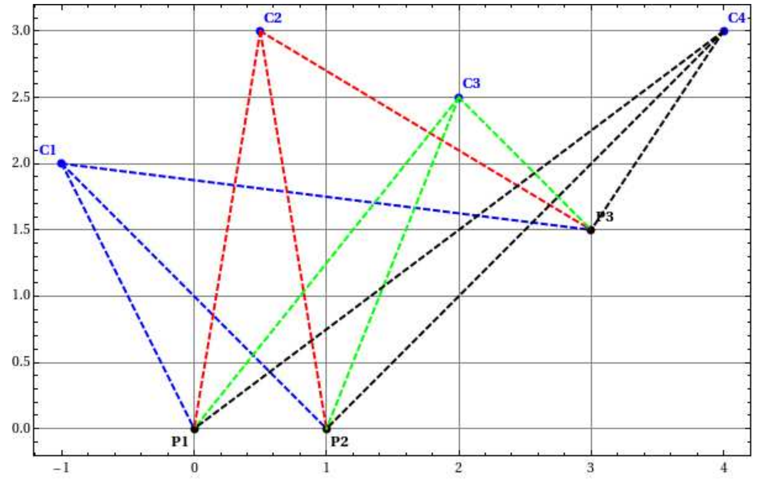


Figure 4.4: Cameras and Points in the plane - More cameras than points

Point positions:

$$P_1 = (0,0)$$

$$P_2 = (1,0)$$

$$P_3 = (3,\frac{3}{2})$$

Camera positions:

$$C_1 = (-1,2)$$

$$C_2 = (\frac{1}{2},3)$$

$$C_3 = (2,\frac{5}{2})$$

$$C_4 = (4,3)$$

The matrix representation of the system of equations is

$$A = \begin{pmatrix}
-\frac{2}{\sqrt{5}} & 0 & 0 & 0 & -\frac{1}{\sqrt{5}} & 0 & 0 & 0 & \frac{2}{\sqrt{5}} & 0 & 0 & \frac{1}{\sqrt{5}} & 0 & 0 \\
0 & -\frac{6}{\sqrt{37}} & 0 & 0 & 0 & \frac{1}{\sqrt{37}} & 0 & 0 & \frac{6}{\sqrt{37}} & 0 & 0 & -\frac{1}{\sqrt{37}} & 0 & 0 \\
0 & 0 & -\frac{5}{\sqrt{41}} & 0 & 0 & 0 & \frac{4}{\sqrt{41}} & 0 & \frac{5}{\sqrt{41}} & 0 & 0 & -\frac{4}{\sqrt{41}} & 0 & 0 \\
0 & 0 & 0 & -\frac{3}{5} & 0 & 0 & 0 & \frac{4}{5} & \frac{3}{5} & 0 & 0 & -\frac{4}{5} & 0 & 0 \\
-\frac{1}{\sqrt{2}} & 0 & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} & 0 \\
0 & -\frac{6}{\sqrt{37}} & 0 & 0 & 0 & -\frac{1}{\sqrt{37}} & 0 & 0 & 0 & \frac{6}{\sqrt{37}} & 0 & 0 & \frac{1}{\sqrt{37}} & 0 \\
0 & 0 & -\frac{5}{\sqrt{29}} & 0 & 0 & 0 & \frac{2}{\sqrt{29}} & 0 & 0 & \frac{5}{\sqrt{29}} & 0 & 0 & -\frac{2}{\sqrt{29}} & 0 \\
0 & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 \\
-\frac{1}{\sqrt{65}} & 0 & 0 & 0 & -\frac{8}{\sqrt{65}} & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{65}} & 0 & 0 & \frac{8}{\sqrt{65}} \\
0 & -\frac{3}{\sqrt{34}} & 0 & 0 & 0 & -\frac{5}{\sqrt{34}} & 0 & 0 & 0 & 0 & \frac{3}{\sqrt{34}} & 0 & 0 & \frac{5}{\sqrt{34}} \\
0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \\
0 & 0 & 0 & -\frac{3}{\sqrt{13}} & 0 & 0 & 0 & \frac{2}{\sqrt{13}} & 0 & 0 & \frac{3}{\sqrt{13}} & 0 & 0 & -\frac{2}{\sqrt{13}} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}$$

with the variables

$$x = \{a(1), a(2), a(3), a(4), b(1), b(2), b(3), b(4), x(1), x(2), x(3), y(1), y(2), y(3)\}$$

$$b = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0\} .$$

4.3 Mathematics of the reconstruction in the 3D case

We follow here [Knill and Ramirez-Herran, 2007b]: For points $P_i = (x_i, y_i, z_i)$ and camera positions $Q_j = (a_j, b_j, c_j)$ in space, the full system of equations for the unknown coordinates is nonlinear. However, we have already solved the problem in the plane and all we need to deal with is another system of linear equations for the third coordinates z_i and c_j.

The slopes $n_{ij} = \cos(\phi_{ij})/\sin(\phi_{ij})$ determine

$$c_i - z_j = n_{ij} r_{ij} ,$$

where $r_{ij} = \sqrt{(x_i - a_j)^2 + (y_i - b_j)^2}$ are the distances from (x_i, y_i) to (a_j, b_j). This leads to a system of equations for the additional unknowns z_i and c_j.

Lets recall the corresponding result for Omnidirectional camera reconstructions in space: the reconstruction of the scene and camera positions in 3D has a unique solution if both the xy-projections of the point configurations as well as the xy-projection of the camera configurations are not collinear and the union of point and camera projections are not contained in the union of two lines [Knill and Ramirez-Herran, 2007b]

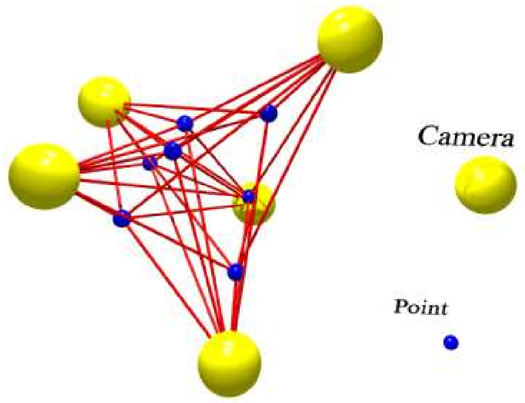


Figure 4.5: The reconstruction in space

In order to get the positions of cameras and scene coordinates, one has only to solve the equations

$$\begin{aligned}
\cos(\theta_{ij})(b_i - y_j) &= \sin(\theta_{ij})(a_i - x_j) \\
\cos(\phi_{ij})(c_i - zj) &= \sin(\phi_{ij}) r_{ij} \\
(x_1, y_1, z_1) &= (0, 0, 0) \\
x_2 &= 1
\end{aligned}$$

for the unknown camera positions (a_i, b_i, c_i) and scene points (x_j, y_j, z_j). First we solve $nm + 3$ equations for the $2n + 2m$ unknowns

$$\begin{aligned}
\cos(\theta_{ij})(b_i - y_j) &= \sin(\theta_{ij})(a_i - x_j) \\
(x_1, y_1) &= (0, 0) \\
x_2 &= 1
\end{aligned}$$

using a least square solution. If we rewrite this system of linear equations as $Ax = b$, then the solution is $x_{min} = (A^T A)^{-1} A^T b$.

Since the reconstruction is unique in the plane, x_i, y_i, a_j, b_j, we can form $r_{ij} = \sqrt{(a_i - x_j)^2 + (b_i - y_j)^2}$ and solve the $n \cdot m + 1$ equations

$$\begin{aligned} \cos(\phi_{ij})(c_i - z_j) &= \sin(\phi_{ij}) r_{ij} \\ z_1 &= 0 \end{aligned}$$

for the additional $n+m$ unknowns. Also this is a least square problem. In case we have two solutions, we have an entire line of solutions. This implies that we can find a deformation $c_i(t), z_j(t)$ for which the angles $\phi_{ij}(t)$ stays constant. Because the xy-differences r_{ij} of the points are known, these fixed angles assure that the height differences $c_i - z_j$ between a camera Q_i and a point P_j is constant. But having $c_i - z_j$ and $c_k - z_j$ constant assures that $c_i - c_k$ is constant too and similarly having $c_i - z_j$ and $c_i - z_k$ fixed assures that $z_j - z_k$ is fixed. In other words, the only ambiguity is a common translation in the z axes, which has been eliminated by assuming $z_1 = 0$. The reconstruction is unique also in three dimensions.

The following is an example to demonstrate the 3D reconstruction described above. The First step is to reconstruct in 2D using the projection of the 3D points onto the xy-plane as input and solving the systems of equations with Least Squares. To simplify the illustration we have built the example on the last 2D example by adding a non-zero z-coordinate to some of the points and some of the cameras leaving the xy-coordinates untouched. The points are the following:

$$p_1 = \{0, 0, 0\}$$

$$p_2 = \{1, 0, 0\}$$

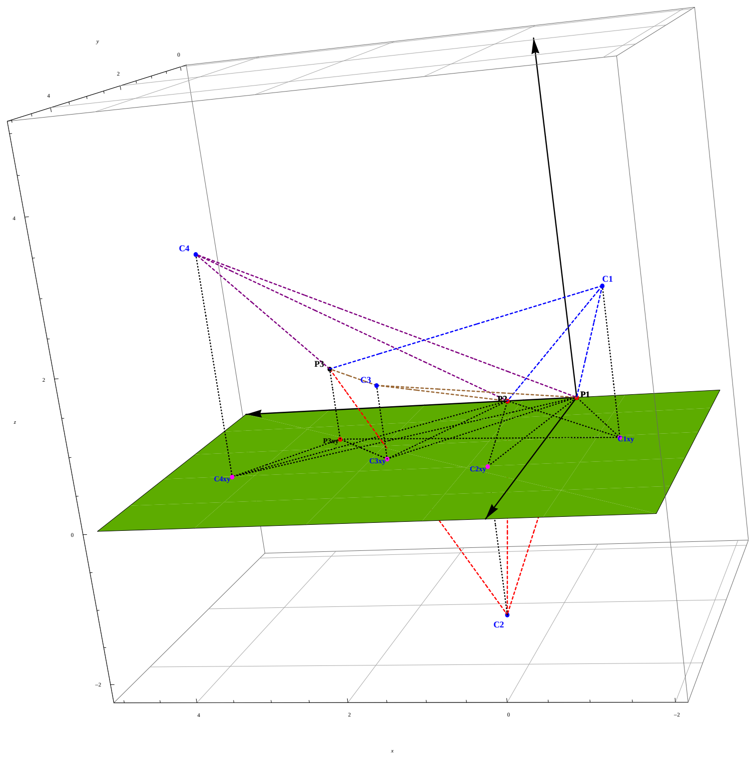


Figure 4.6: Example of the reconstruction in space showing the x-y proyections

$$p_3 = \left\{3, \frac{3}{2}, 1\right\}$$

And the cameras are located as follows:

$$cam_1 = (-1, 2, 2)$$

$$cam_2 = (\frac{1}{2}, 3, -2)$$

$$cam_3 = (2, \frac{5}{2}, 1)$$

$$cam_4 = (4, 3, 3)$$

The second step is to reconstruct the heights, "the z-coordinate by using Least squares again", as it was described above. This means that we assume at this point that we got the 2D points

$$xy - coords = \left\{-1, \frac{1}{2}, 2, 4, 2, 3, \frac{5}{2}, 3, 0, 1, 3, 0, 0, \frac{3}{2}\right\}$$

corresponding to the variables

$$x = \{a(1), a(2), a(3), a(4), b(1), b(2), b(3), b(4), x(1), x(2), x(3), y(1), y(2), y(3)\}$$

and we can use this to find the distances rij and calculate the angles. The matrix repre-

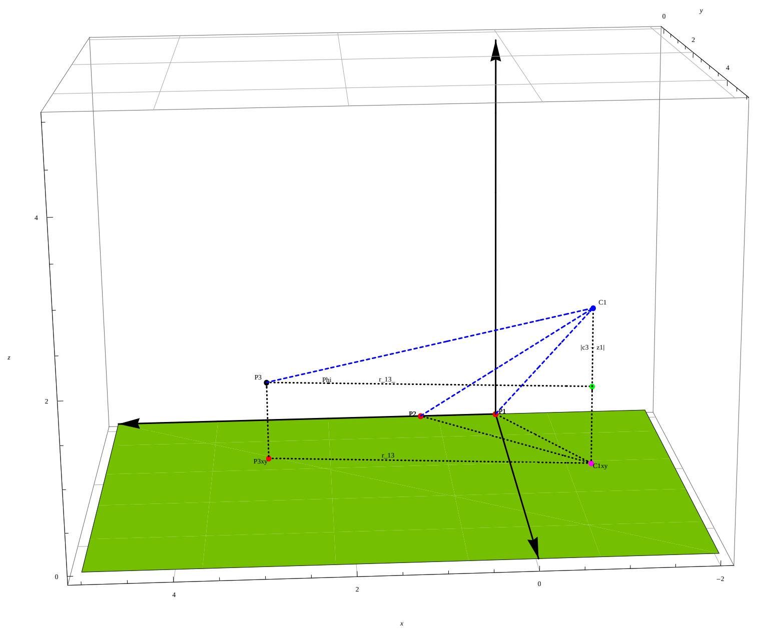


Figure 4.7: The reconstruction in 3D space

sentation of the system of equations on the variables c1, c2, c3, c4, corresponding to the z-coordinates of the camera positions, and z1, z2, z3, corresponding to the z-coordinates of the points to be reconstructed, is

$$AZ = \begin{pmatrix} 0.745356 & 0 & 0 & 0 & -0.745356 & 0 & 0 \\ 0 & 0.835532 & 0 & 0 & -0.835532 & 0 & 0 \\ 0 & 0 & 0.954521 & 0 & -0.954521 & 0 & 0 \\ 0 & 0 & 0 & 0.857493 & -0.857493 & 0 & 0 \\ 0.816497 & 0 & 0 & 0 & 0 & -0.816497 & 0 \\ 0 & 0.835532 & 0 & 0 & 0 & -0.835532 & 0 \\ 0 & 0 & 0.937437 & 0 & 0 & -0.937437 & 0 \\ 0 & 0 & 0 & 0.816497 & 0 & -0.816497 & 0 \\ 0.970582 & 0 & 0 & 0 & 0 & 0 & -0.970582 \\ 0 & 0.696932 & 0 & 0 & 0 & 0 & -0.696932 \\ 0 & 0 & 1. & 0 & 0 & 0 & -1. \\ 0 & 0 & 0 & 0.669534 & 0 & 0 & -0.669534 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

The vector b correspond to the constant rij Sin * Phi,ij in the equations and for this case is:

$$bz = \{1.49, -1.67, 0.95, 2.57, 1.63, -1.67, 0.937, 2.45, 0.97, -2.09, 0, 1.34, 0\}$$

After using Least Squares with input AZ and bz we got the following z-coordinate reconstruction:

$$z = \{2., -2., 1., 3., 0, 0, 1.\}$$

where the variables corresponding in the array are

$$z - coords = \{c(1), c(2), c(3), c(4), z(1), z(2), z(3)\}$$

the final result is the set of point coordinates in 3D corresponding to the geometry seen by the Omnidirectional camera and the positions of the cameras as coded in the xy-coords,z-coords arrays.

4.4 Error Estimation

How does the reconstructed scene depend on measurement or computation errors? How sensible is the least square solution on the entries of A? The error depends the volume $\sqrt{\det(A^T A)}$ of the parallelepiped spanned by the columns of A which form a basis in the image of A. Because this parallelepiped has positive volume if we are in a non-ambiguous situation, we have:

The maximal error ϵ of the reconstruction depends linearly on the error δ of the angles θ_{ij} and ϕ_{ij} for small δ. There exists a constant C such that $|\epsilon(\delta)| \leq C|\delta|$.

The reconstruction problem is a least square problem $Ax = b$ which has the solution $x_* = (A^T A)^{-1} A^T b$. Without error, there exists a unique solution. In general, A has no kernel if we are not in an ambiguous situation. The error constant depends on the maximal entries of A and $C = 1/\det(A^T A^{-1})$ which are both finite if A has no kernel.

Empirically [Knill and Ramirez-Herran, 2007b], we confirmed that the maximal error is of the order of δ. We only made experiments with synthetic but random data and see that the constant C is quite small. The computer generated random points, photographs them with an Omnidirectional cameras from different locations and reconstructs the point locations from the angle data. The maximal error is expected to grow for a larger number of points because the length of the error vector grows with the dimension. A random vector $\vec{\delta} = (\delta_1, \dots, \delta_n)$ with δ_i of the order δ has length of the order $\sqrt{n}\delta$.

Remarks:

1. The average error becomes smaller if n is larger.

2. From the practical point of view, we are also interested in how much aberration we see when the reconstructed scene is filmed again. Geometrically, the least square solution x_* of the system $Ax = b$ has the property that Ax_* is the point in the image of A which is closest to b. If the reconstructed scene is filmed again, then even with some errors, the camera sees a similar scene. Because $A(A^T A)^{-1} A^T$ is a projection onto the image of A, this projected error is of the order 1. In other words, we see no larger errors than the actual errors. For refined error estimates for least square solutions see [Wei, 1989, Golub and Van Loan, 1980].

Chapter 5

Tracking and 3D reconstruction

This chapter is dedicated to the tracking of the features extracted using our corner detector across multiples frames in both real and synthetic movies. We explain here our tracking strategies, and we give samples of the results of our experiments.

5.1 The correspondence problem.

The correspondence problem is an important step needed to do the reconstruction. It is the problem to identify corresponding points on different frames of a movie. This is an engineering task were one can take advantage of using low-level feature extraction techniques explained in section 2.4.2. It is important to mention here that our main focus is in the structure from motion problem and that we don't attempt to produce new results in the solution of the correspondences.

Let $A_0, A_1, ..., A_m$ be picture frames, as shown in figure above, so that A_0 corresponds to time t_0 and A_m corresponds to time t_m. The task is to identify for each picture A_i, n points $x_{i,m}$ such that the points $x_{i,j} = \{j = 0, ..., m\}$ correspond. In a second stage, we

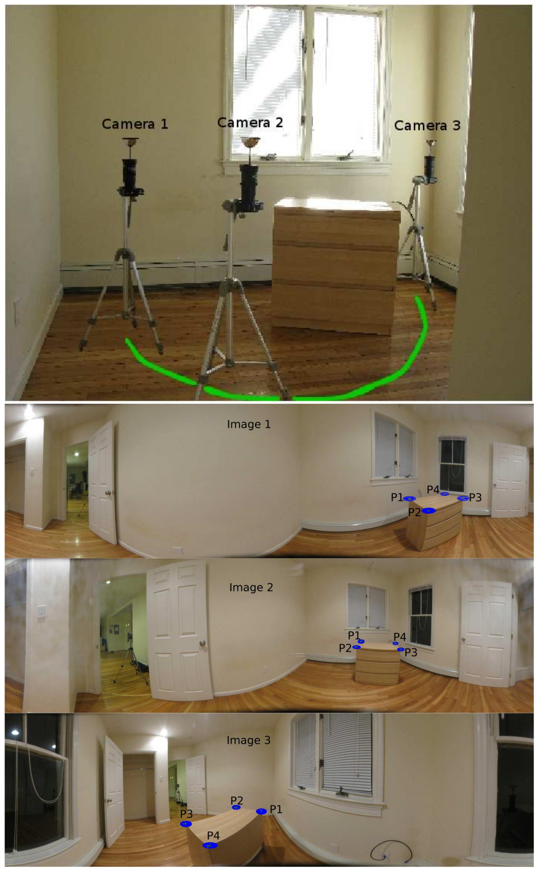


Figure 5.1: Point correspondences in panoramic pictures

want to construct parameterized curves $x(i, t)$ in the plane which describe the motion so that $x(i, j) = x(i, j/m)$. Here is an outline of the correspondence problem process:

1. Identifying important points. To do this, we filter the pictures to see important object features using skeletonization, taking boundaries. Then we extract the interesting points.

2. Match points in neighboring pictures. Because the frames are close, we have to search only in a small neighborhood. The neighborhood can be narrowed by estimating the velocity of the point from previous pictures. To find the point y in a picture B matching

a point x in picture A, we find the pixel (y_1, y_2) in a neighborhood of the location (x_1, x_2) which minimizes

$$L(y_1, y_2) = \sum_{i,j} (A(x_1 + i, x_2 + j) - B(y_1 + i, y_2 + j))^2$$

where $A(i, j) = (r, g, b)$ is the color vector at pixel (i, j) and where the sum is over a rectangle of fixed size.

3. Interpolate over larger time frames. In a real situation, points disappear behind other objects and are invisible for some time. Longer interpolations allow tracking points also when they are hidden. To do so, we delete corresponding points, if there distance becomes too large. Then we fill in the missing points by interpolating the remaining points using curve fitting. Previous work on this is the IPAN tracker and KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker Lucas, B.D. and Kanade, T.(1991).

5.2 Tracking points in synthetic scenes.

In general the matching is based in similarity of appearance and it uses a window of certain size to look for possible matches in frame n+1 in the neighborhood of the corner in frame n. Most of the matching algorithms assume that the changes between images are not big and that the physical entities look the same between pictures. This assumption is one of the main reasons why feature tracking is still a fundamentally hard problem to solve because there is a lot of phenomena that can happen between frames such as changes in illumination and occlusion.

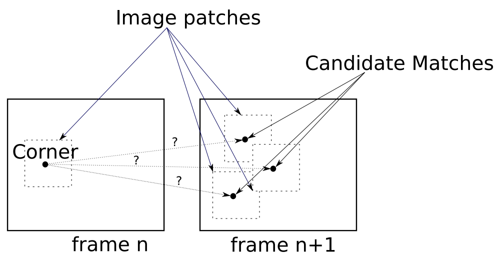


Figure 5.2: Basic matching

5.2.1 KLT Corner Detector with tracking

KLT is another method to solve correspondences and track features. This method defines the measure of match between fixed-size feature windows in the past and current frame as the sum of squared intensity differences over the windows. The displacement is then defined as the one that minimizes this sum.

KLT is an abbreviation for **”Kanade-Lucas-Tomasi Feature Tracking**. It is implemented in the C programming language. The source code is in the public domain, available for both commercial and non-commercial use.

The tracker is based on the early work of Lucas and Kanade [Lucas and Kanade, 1981] and was developed fully in [Lucas and Kanade, 1991], and was explained in more detail in the work of [Shi and Tomasi, 1994]. Tomasi proposed a slight modification, which makes the computation symmetric with respect to the two images - the resulting equation is derived in the unpublished note by Birchfield [S.Birchfeld, 1997].

Good features are located by examining the minimum eigenvalue of each 2 by 2 gradient matrix, and features are tracked using a Newton-Rapson method of minimizing the difference between the two windows. Multi-resolution tracking allows for relatively large displacements between images.

In this case the images are converted to gray level images before extracting the interesting points. In the following figure we show the points marked in red/gray in small size. Notice that KLT does not choose what we expect to see as corners in all the cases. KLT is also ignoring the intersection of the lines in the checker texture of the boxes and don't use them as interesting points. In the following figure we show the merged frames used

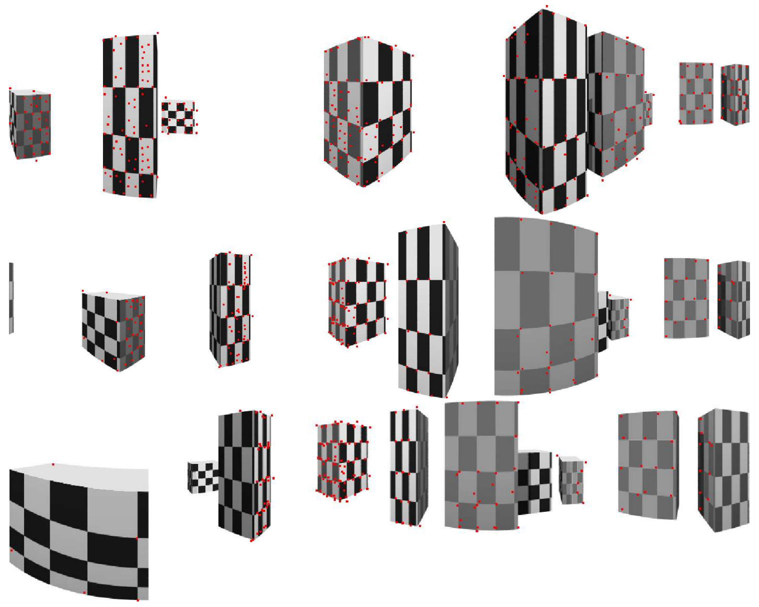


Figure 5.3: Tracking points on each frame using KLT

in the tracking the interesting points. Merging the pictures is used to visualize motion in this case.

We do not use here the KLT corner detection but look for features in a single frame, then track them over some time.

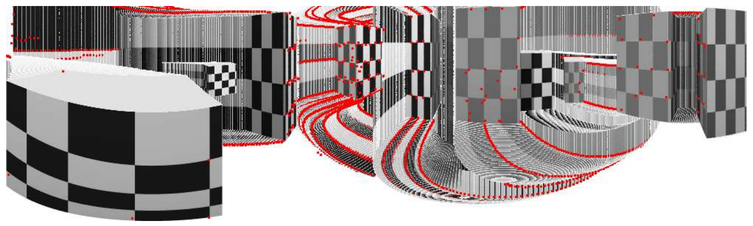


Figure 5.4: Trajectory of the tracked points using KLT

5.2.2 OMNIVIS Corner Detector with tracking

In the following figure we show results of tracking based in the matching across pictures using the Kitchen-Rosenfeld corner detector and thresholds between pictures.

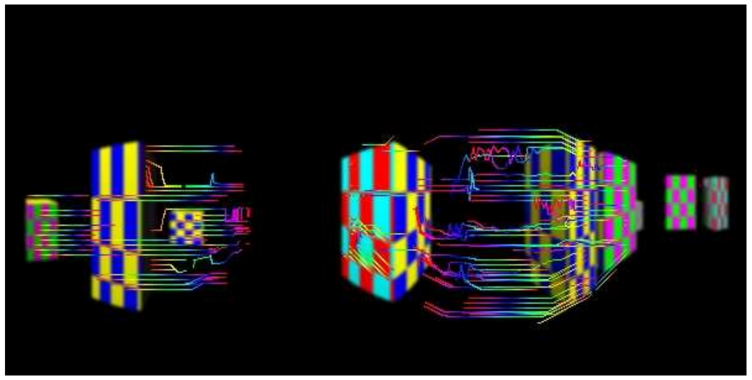


Figure 5.5: Tracking points in a synthetic movie

Tracking.

In the case of a movie, where several pictures are involved, one has first of all to use the just discussed low and high level features. There is then also the problem to match the features which have been found in different frames. This is called **feature**

tracking. This requires techniques that track motion and is generally called optical flow. Feature tracking over several frames can help to understand high level features in individual frames. Structure from motion algorithms try to reconstruct the physical scene from the movie information.

5.3 Tracking strategy

There are different strategies to track features. Points can be tracked for some time, but usually lose their particularities over time. A corner for example can become occluded. Even if it stays visible, it can lose the corner property over time. Instead of adding new points as old ones are lose, we restart the entire feature tracking process periodically at some key frames K_1, K_2, K_3. We make the tracking intervals overlap. Since we are using least square fitting methods, we do not insist that a point can be tracked over the entire time span. This is also not realistic. When drive through a town, one can follow a particular point only for a certain period of time. The parameters are the tracking interval length and the key frame times. In the following figure we show intervals in which a particular corner was tracked. We show the keyframes k_i in which the corner start to be tracked.

In the next figure we show how the situation happen withh multiple points, illustrating how different points can be tracked through different time intervals.

A C program computes these tracked points and exports them as a text file. Mathematica picks these points up and does the reconstruction. The format in which these point are stored is show iin the following figure.

There is a Mathematica program called translate.m, which takes these data in "points.dat" generated from the C program and produces Mathematica data in a file "out.m" which

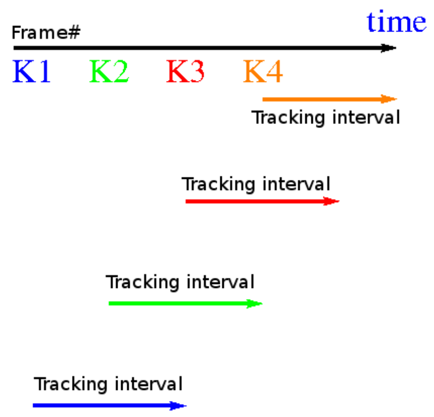


Figure 5.6: Tracking points by intervals in a movie

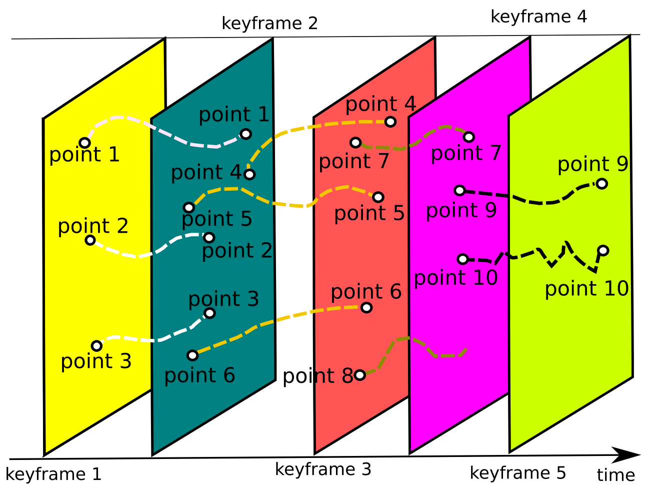


Figure 5.7: Keyframes determine intervals in which the tracking is good

can be understood by the reconstruction program "statistics.m". The data out.m only contain point data from a few camara points.

Additionally we experimented with tracking using multiple levels of Gaussian smoothing to compare the different results of the tracker for different smothing levels and visually

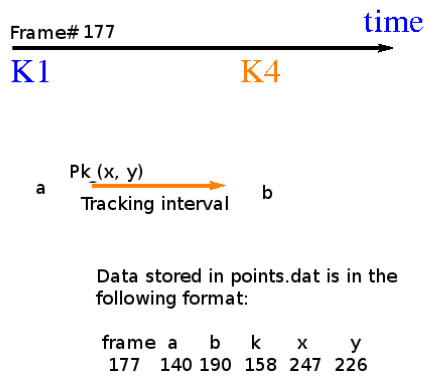


Figure 5.8: Coding the results of tracking using our strategy

decide which levels of smoothing were more convenient in terms of getting not too many points and that the points could be tracked through enough number of pictures.

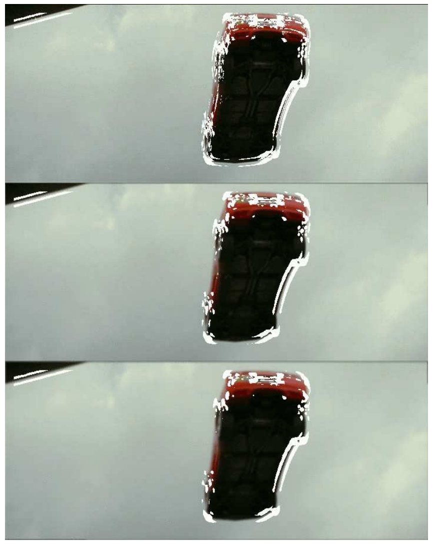


Figure 5.9: Experimenting with multiple Gaussian smoothing for tracking

Later on the project we decide to do statistics of the intensities, gradients, curvatures

and cross values to have a sense which could be the values for the thresholds before running the program. The list of parameters to determine for running the program are in the following list.

Tuning Parameters:

- Gaussian averaging
- Grid size
- Gradient, curvature and cross thresholds
- Number of frames
- Tracing span
- Frame rate

Automatic determination of the threshold parameter

We determine the thresholds by process the first picture to determine the maximal and minimum values of the following parameters:

- curvature values for each channel (red, green, and blue)
- gradient length for each channel (red, green, and blue)

Once the extreme values are calculated, we map these minimum and maximum values for curvature and gradient for each color,to a scale between 0 and 255. Notice that all the values are positive quantities. Scaling the values between 0 and 255 allow us to produce pictures of the curvatures and gradients as the one shown below.

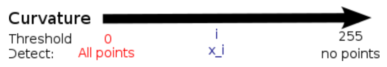


Figure 5.10: Curvature is transformed in color levels for display and storage

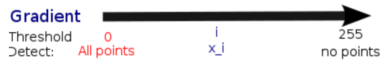


Figure 5.11: Gradient is transformed in color levels for display and storage

The purpose of this strategy is two folded: to store the information about curvature/gradient of each picture to be used in posterior calculations and to establish a threshold automatically.

As we shown in the last two figures, the treshold can now be setup depending on the situation inside the interval between 0 and 255.

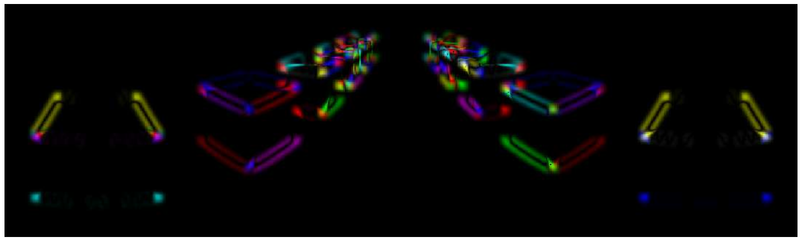


Figure 5.12: Curvature threshold calculation in the first picture

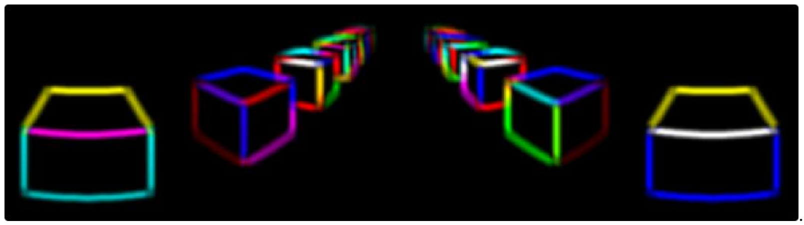


Figure 5.13: Gradient threshold calculation in the first picture

5.4 Tracking corners in synthetic scenes

We used the POV-Ray language to create three-dimensional, photo-realistic images that we will use later for reconstruction. This task involve to chose different types of 3D objects, colors, the dimension of the objects, textures, and camera paths in such a way that covers most of the possible configurations.

For our experiments, in the simplest case, we use boxes to represent buildings. In the real scenes we will be looking to make video for capturing this kind of scenes at different scales. The reconstruction is not measured for the complexity of the shapes used in the scenes but for quantities related with the performance of the optimization methods in which the reconstruction is based. The figure shows an example of a simple scene rendered in POV-Ray.

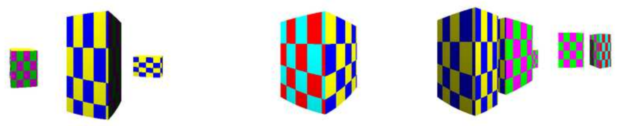


Figure 5.14: A scene of cubes rendered in POV-Ray

In the two following scenes, the white points are use to show the interested points and they are tracked across multiple frames. The image is smoothed to make easier the matching using thresholds.

The last figure after the sequence, shows the trajectories of the points tracked using white lines drawn on top of the blocks. Scene 2 shows another sequence of pictures from a scene with blocks that have a texture type checker. Notice that there are interesting points on the intersection of lines in the faces of the blocks. The last figure after the

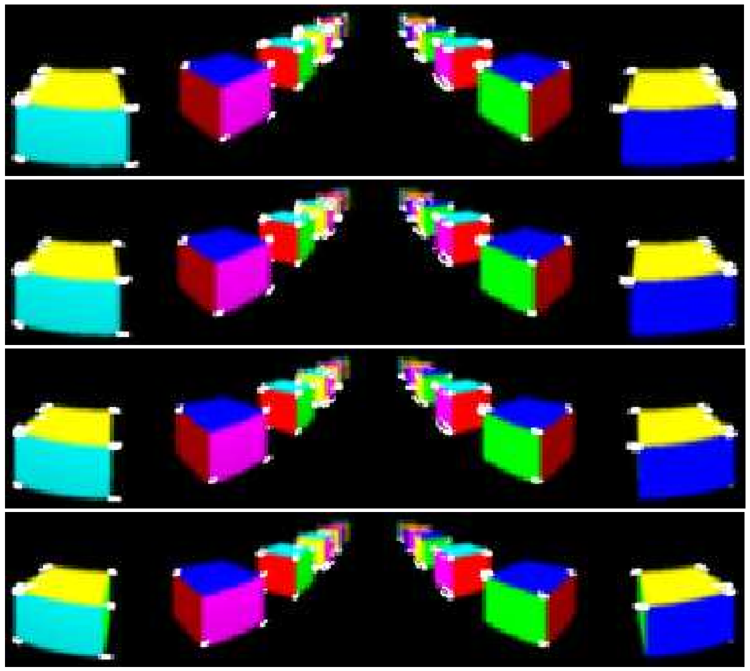


Figure 5.15: Sequence of pictures with tracked points - scene 1

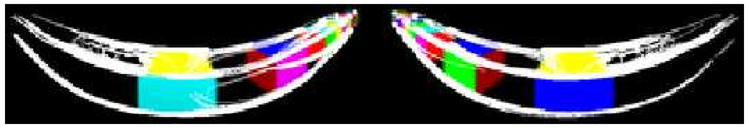


Figure 5.16: The tracked points - scene 1

sequence, shows the trajectories of the points tracked using white lines drawn on top of the blocks. This is a way to represent the motion.

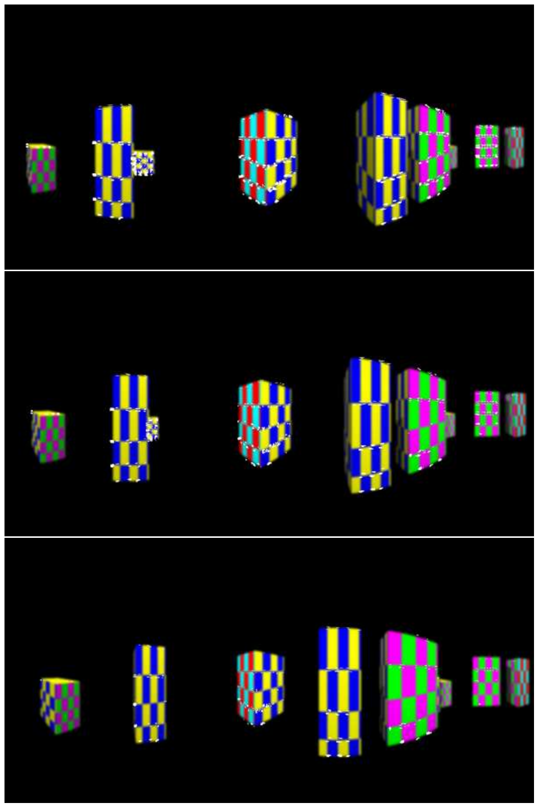


Figure 5.17: Sequence of pictures with tracked points - scene 2

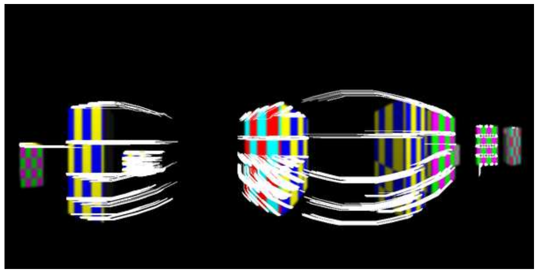


Figure 5.18: The tracked points - scene 2

5.5 Tracking corners in real scenes

We collected a number of video streams from different places and we used the tracking and produced the results shown in the next sections of the chapter.

5.5.1 Florida movie

We collected a number of video streams in Palm Beach Florida. For capturing the video we set up a 360 camera in a tripod in top of a car.

POINTS OF INTEREST:
1 Pan's Garden
2 Worth Avenue
3 Major Alley
4 Society of the Four Arts
5 Lake Trail
6 Sea Gull Cottage
7 The Flagler Museum
8 Royal Poinciana Chapel
9 Bethesda-by-the-Sea
10 Phipps Plaza Historic District
11 Memorial Fountain
12 Town Hall
Palm Beach Chamber of Commerce
Royal Poinciana Way Shopping District
South County Road Shopping District
Worth Avenue Shopping District

Figure 5.19: Map of Palm Beach, FL

We extracted pictures from selected portions of the video an unwrapped the pictures to be used for the experiments. In this data set we found the most noise. A lot of vibration of the camera produces an erratic motion the light and humidity also affected the movie. Sometimes there is so much a motion, blur and light changes that it is difficult to predict

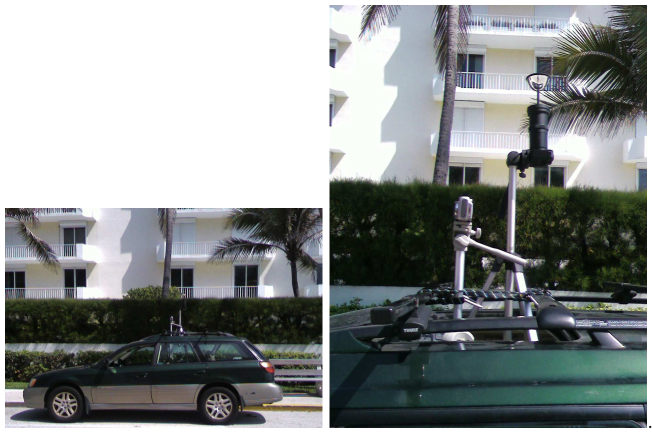


Figure 5.20: OmniCamera mounted on top of a Subaru for omnivideo capture

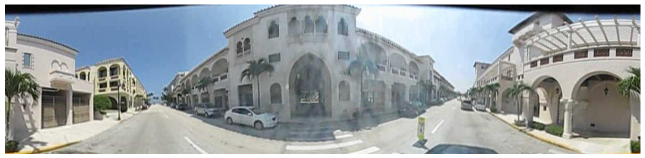


Figure 5.21: Panorama - Palm Beach, FL

which parameters to use to get good tracking. In the figure we show three frames for each of the three different levels of smoothing. We can notice how the smoothing affects the number of interesting points detected to be used for tracking. The following figure shows the tracking using red lines that show the trajectories of the points tracked.

In the next figure, we show four pictures: the original, the picture of curvature, and gradient, and finally the smoothed picture. The program use pictures to store this measurements in a temporary directory. In case of wanted to run the program again, these measurements are already done and then it reduces the time needed for doing the tracking.

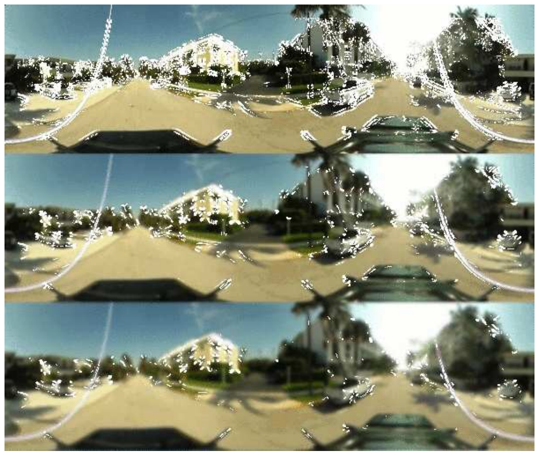


Figure 5.22: Tracking corners in multiple smoothing levels - Palm Beach, FL

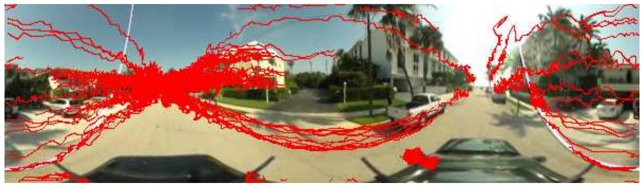


Figure 5.23: Trajectories of tracked points. Palm Beach, FL

5.5.2 Harvard University Science Center movies

We ran the program to track the corners in a corridor inside the Math department at Harvard. Here we show the results. We use different levels of smoothing and notice how the gradients are bigger in the more smoothed pictures . From top to botton the pictures are more smoothed.

In another scene we capture the piano room at the math department at Harvard. Here we show the results. Again there are three sets of three pictures and the smoothing increases from top to bottom. The gradients are bigger at the bottom and that is why they show bigger lines representing bigger gradients. The threshold is bigger for pictures

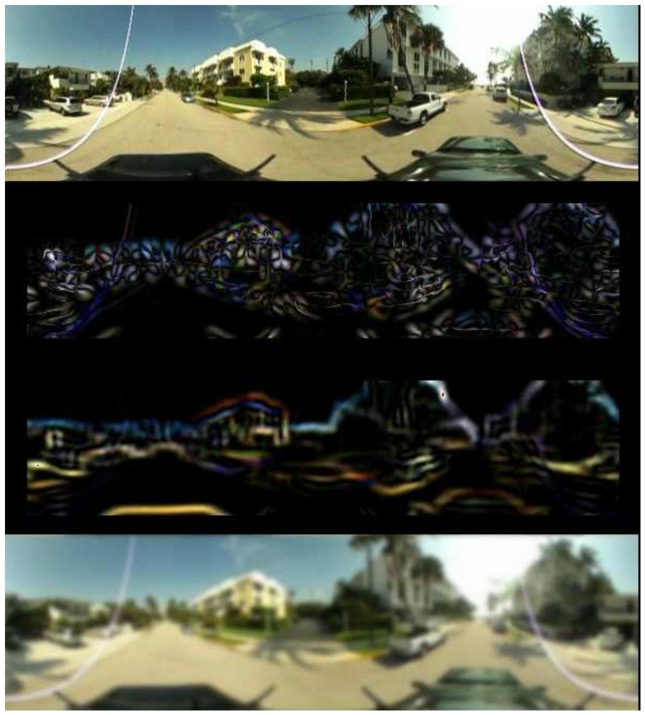


Figure 5.24: Original, Curvature, gradient and Gaussian smoothing pictures - Palm Beach, FL

that have more smoothing.

5.5.3 Rotating a Rubik cube movie

We experimented also with a Rubik cube on a rotating platform. While we could track the points that are part of the rug, this points are not the more interesting. In this case the camera does not move. In a scene like this, we can determine to find and track only the most interesting features by limiting the processing only to a region of the picture. For example if the idea is to reconstruct only the Rubik's cube we can analyze

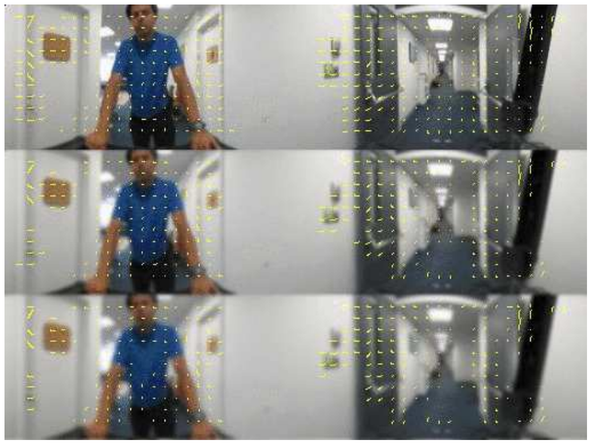


Figure 5.25: Tracking corners with multiple smoothing - Harvard Math department

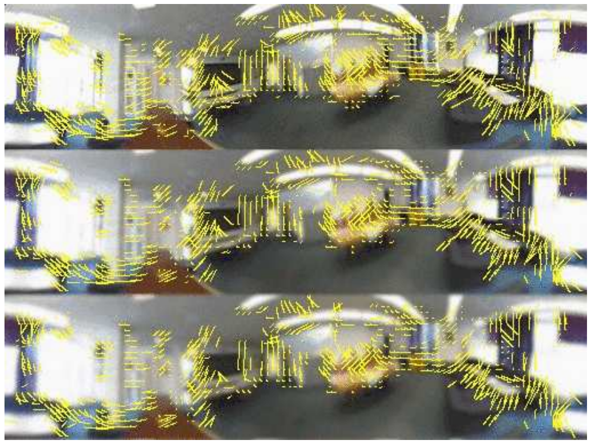


Figure 5.26: Tracking corners with multiple smoothing - Harvard Math department

the Rubik's cube bounding box by extracting the edges from the first picture and finding the rectangle that fits, then we could circumscribe a circle on it. And allow some tolerance in the measurements to account for variations across pictures produced by the rotational motion. Another way to leave this features out could be by thresholding the blue intensity by using segmentation.

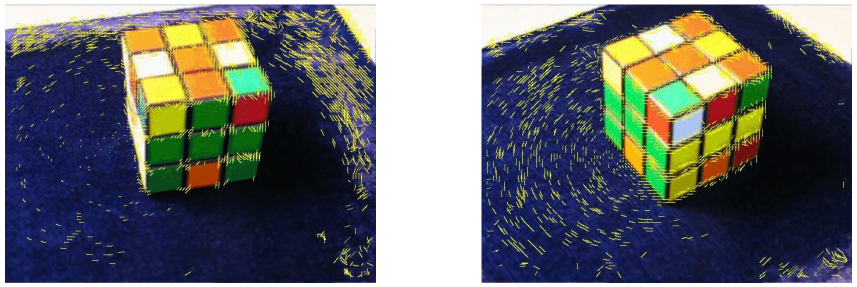


Figure 5.27: Tracking corners in a rotating Rubik cube

5.5.4 Ballons Movie

We also experimented in tracking large movements such as in a movie where there are balloons flying. In this case we demostrated that the tracking program implemented can also track objects that move considerably. Inside the code in the appendix we have a parameter that fix the searching distance for the algorithm to find a match. We tested our tracking with several sequences of this movie getting results such as the one we show in the following figure.

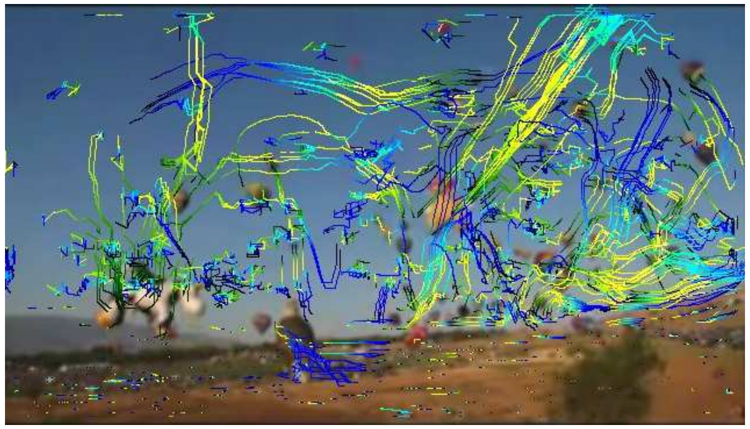


Figure 5.28: The tracking algorithm in balloons shows the path for each corner

5.5.5 Robot taking a movie of a small scene

We decided to build a small robot using a NXT controller and Lego pieces. The idea is to create a real scene using simple objects and perform the tracking.

In the following figure we show the results of tracking the corners. We noticed that there

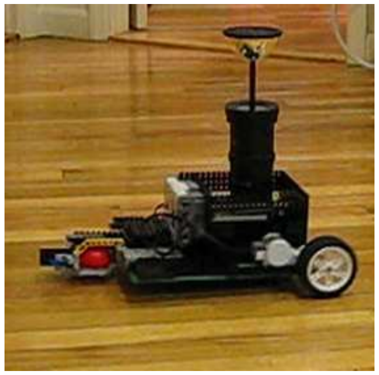


Figure 5.29: Robot *OmniRobot* using Lego NXT controller.

were sudden motions of the camera. Given that this motion or noise could be inevitable in any real scene taken, we implemented a filter based in the measurement of the variations of the point coordinates. This allows us to reduce the number of points by substracting the points with large variations from the set.

5.6 3D Reconstruction for Omnidirectional vision

The following figure shows the pipeline used for both reconstruction of real and synthetic scenes. In this section we show the results of a reconstruction on a synthetic scene created in POV-Ray.

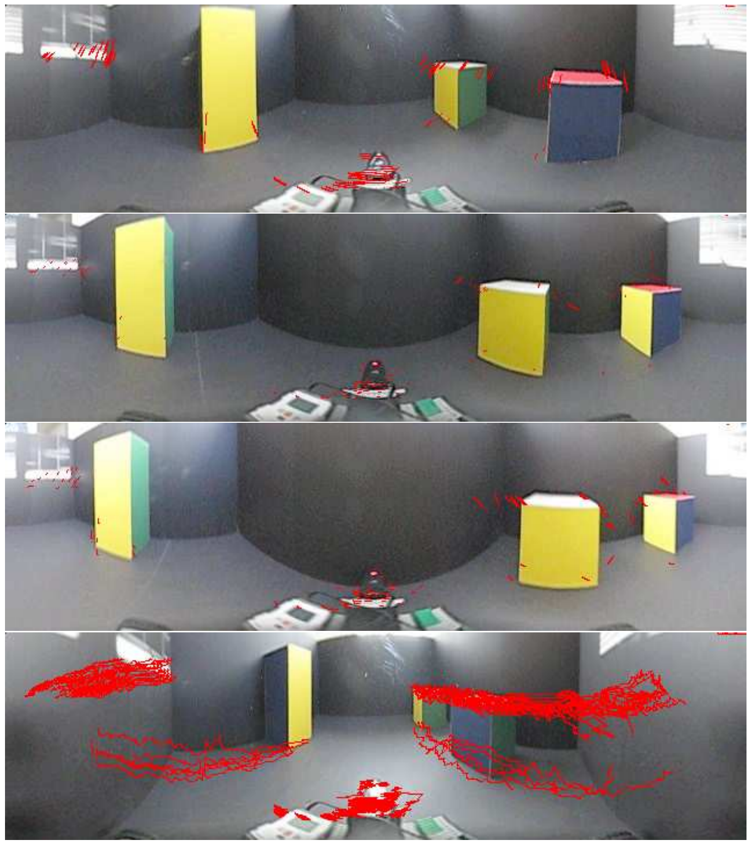


Figure 5.30: Tracking corners on real blocks

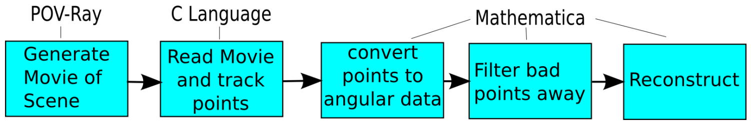


Figure 5.31: Implemented Pipeline of the 3D reconstruction

5.6.1 3D Reconstruction in a synthetic scene

We performed a reconstruction of the synthetic scene by getting the tracked points manually. The following figure show the starting frame of the movie in which we based the

3D reconstruction of a Synthetic scene made of cubes.

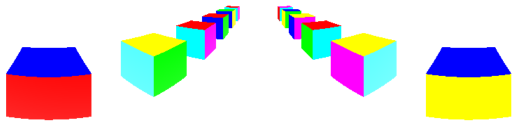


Figure 5.32: Synthetic scene to be reconstructed

In the following figure we show all the points and frames together. This image shows the path of each point in a diferent color.

Figure 5.33: Trajectory of the points to be reconstructed

There are two figures in the appendix called: Input data for the reconstruction of the synthetic scene. These figures show the data used for the reconstruction. The points were marked manually using an image editing program called GIMP. The first figure shows the first frame of the movie, and the second show the last frame from a movie containing 100 images from which we only selected 10 frames and tracked 10 points.

The file points.dat contain the pixel coordinates of the points tracked and shown in the appendix called: Input data for the reconstruction of the synthetic scene.
The following is an example of a list of points stored in the points.dat file.

```
0 0 0 0 1200 600
3 3 12 3 937 378
3 3 12 4 960 415
3 3 12 5 960 476
3 3 12 6 839 476
3 3 12 7 839 415
3 3 12 8 862 378
3 3 12 9 337 378
3 3 12 10 360 415
```

```
3 3 12 11 360 477
3 3 12 12 239 476
12 3 12 3 1139 327
12 3 12 4 1160 328
12 3 12 5 1160 352
12 3 12 6 1155 359
12 3 12 7 1155 332
12 3 12 8 1127 330
12 3 12 9 72 330
12 3 12 10 45 331
12 3 12 11 44 359
12 3 12 12 39 352
```

The first line has the information about the size of the images. The remaining lines give the information about the frame number, the interval in which the point was tracked, and the x y pixel coordinates of the point. Notice that we only show the data corresponding to the first and ast frame.

The program translate.m transform this data into angular data that can be used in the reconstruction. The following output is an example of the data stored in out.dat after points.dat has been translated.

```
Obs={{4.90612,5.17839,5.38783,5.53968,5.67057,5.75435,5.82242,5.88001,5.93237,5.96379},
{5.02655,5.3983,5.62869,5.78053,5.86431,5.93237,5.9795,6.01615,6.04757,6.07375},
{5.02655,5.40354,5.63392,5.78053,5.87478,5.93761,5.98473,6.02139,6.0528,6.07375},
{4.39299,4.87994,5.29882,5.57109,5.73864,5.8486,5.9219,5.97426,6.01615,6.04757},
{4.39299,4.87994,5.29882,5.57109,5.73864,5.8486,5.9219,5.97426,6.01615,6.04757},
{4.51342,4.81187,5.09462,5.325,5.49779,5.62869,5.72293,5.79624,5.85907,5.90096},
{1.76453,1.46608,1.18333,0.95295,0.780162,0.649262,0.539307,0.481711,0.418879,0.376991},
{1.88496,1.40324,0.97913,0.706858,0.544543,0.434587,0.361283,0.303687,0.267035,0.235619},
{1.88496,1.40324,0.97913,0.706858,0.539307,0.434587,0.361283,0.303687,0.261799,0.230383},
{1.2514,0.87441,0.644026,0.497419,0.403171,0.340339,0.293215,0.256563,0.225147,0.204204}};
```

The 3D data

Now the angular data is ready and we read this data in our mathematica implementation of structure from motion from video taken with an Omnidirectional camera. After running the mathematica program for the reconstruction (statistics.m), t he output file give us the coordinates of the 3D points. In the following list, we show a small sample of the 3D data.

```
P = {{0, 0, 0}, {1, 0, 0.328396796337526}, {0.5049615991833604,
      1.7212487415940303, 2.353005705702351}, {-1.7698782069730215,
      1.797663022110688, 2.264296871744245}, {-1.7698782069730505,
      1.7976630221106844, 0.12734801826514966}, {-1.7625284467976081,
      -0.38119619954099115, 0.12914850308579556}, {-1.7261187219165308,
      10.505938132634814, 0.08717147599523385}, {-1.7240268291144032,
      8.408653369456973, 0.11387681698949685}, {-1.7322288054866701,
      8.378628313158153, 2.2487984691282903}, {0.390007811375965,
      8.370132789926995, 2.258434633986024}}

Q = {{-0.6562036790746708, 5.241759064860666, -2.320238657575981},
     {-2.3137219785555376, 5.1438191036287755, -2.307296377405793},
     {-3.953984281764376, 5.101227085239544, -2.2907949577584596},
     {-5.592659364195745, 5.093708983680959, -2.2853656520176813},
     {-7.2236348431457, 5.090486897540604, -2.2762081760857753},
     {-8.851642798954806, 5.090629278506059, -2.2627120154215152},
     {-10.519537974246688, 5.113985139481018, -2.2823638971081506},
     {-12.118615705274015, 5.10445943320202, -2.2660278344397407},
     {-13.90288033772486, 5.092085846906637, -2.2678696313613513},
     {-15.488734115888953, 5.090340564185675, -2.239680354943926}}
```

We use this 3D data to plot the coordinate of the points resulted from the reconstruction. In the list above P denotes array of points and Q denote array of cameras. The following figure show the points reconstructed in 3D and the motion of the camera is painted in red showing the positions of the camera reconstructed in 3D. We demostrated here that our reconstruction worked well with the synthetic movies up to a scale.

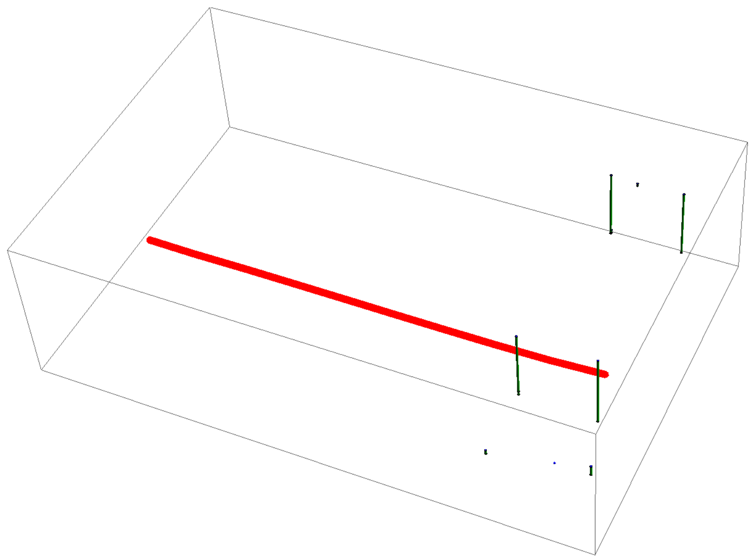


Figure 5.34: Structure and camera reconstruction of a synthetic scene

Chapter 6

Software implementation

This chapter gives details of both functional and non-fuctional requierements as initially proposed for this project. We discuss the results and the verification of this requierements in the next chapter.

6.1 Functional Requirements

The following is a list of the functional requierements of OMNIVIS.

- The system performs a low-level feature extraction from a sequence of images.
- The progam solve the correspondences on the extracted features and store this information so it can be used for the reconstruction.
- The software implemented solve the structure from motion problem from the correspondences, and stores the estimation of both structure and camera motion that will be used in rendering the 3D reconstruction

6.2 Non-functional Requirements

The following is a list of the non-functional requierements of OMNIVIS.

- Documentation: The source code is annotated.
- Efficiency: The system provides a measurement of the resource consumptions (memory and processor use) for the given load (image sequence).
- Effectiveness: the system provides measurements of the performance in relation to effort in terms of number of iterations/recursions for different problem sizes.
- Extensibility: the object oriented programming approach will provide ease of adding features and carry-forward customization in a next major version upgrade.
- Legal and licensing issues: code is released as Open Source. Any platform with a Gnu compiler: Linux OSX, Solaris, PC with Cygwin should work.
- Testability: the system includes a set of testing cases in which the system perform efficiently and provides documented counterexamples.
- The system has a way to validate input that it is going to be used for testing.
- The program provide understandable error messages.

6.3 Development Environment

The development environment for this project will be relatively simple. Coding is carried out in the C language and Mathematica 7 as well as Perl for scripting. We use the open source ray tracer POV-Ray to create scenes and shell scripting to automatize. The nature of all this tools is cross-platform and we will be using the tools in different operating system environments interchangeably.

This project requires a set of software tools which we will develop by dividing the problem in pieces. According with the pipeline we will build application modules for feature extraction and feature tracking, correspondence and the SFM reconstruction.

6.4 Programming Languages

C, Perl, POV-Ray, MathLab, and Mathematica. A summary of the use that we will give to every language and software application is shown in Table 3.

6.5 Implementation

Our objective is to develop new methods and use existing mathematical models to address this specific task: the 3D Space and Camera Path Reconstruction for Omnidirectional Vision. As a starting point we will be using results presented in three papers Ramirez-Herran, J. and Knill, O., 2006 that we wrote in 2006 with Prof. Oliver Knill from the Mathematics department at Harvard in which the reconstruction was perform robustly using synthetic points assuming the correspondences of the points through all the frames were solved. We will also adopt the recommendations given by Oliensis, J. in 2000 for this kind of research. Doing the reconstruction in the real world is even more challenging.

Here is the original pipeline proposed that automatically obtains a detailed 3D model from an image sequence acquired both synthetically and with an Omnidirectional sensor: In the pipeline shown above, we can identify that there are three main problems to be

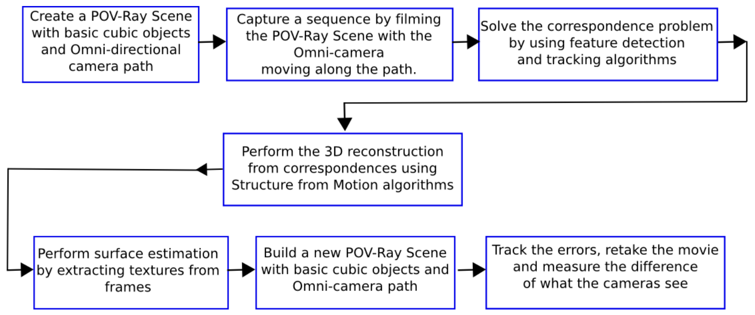

Create a POV-Ray Scene with basic cubic objects and Omni-directional camera path
Capture a sequence by filming the POV-Ray Scene with the Omni-camera moving along the path.
Solve the correspondence problem by using feature detection and tracking algorithms
Perform the 3D reconstruction from correspondences using Structure from Motion algorithms
Perform surface estimation by extracting textures from frames
Build a new POV-Ray Scene with basic cubic objects and Omni-camera path
Track the errors, retake the movie and measure the difference of what the cameras see

Figure 6.1: Proposed 3D Space and panoramic Camera recovery pipeline

treated separated: feature extraction, correspondences and the space and camera path reconstruction (structure from motion). In the following chapters we describe details of the approaches to solve these problems. The software application OMNIVIS amalgamates a series of software tools to achieve the task of reconstruction. We will use the information of the requirements and implementation exposed in this chapter for helping us in the discussion about conclussions, lessons learned and future work that we do in the next chapter.

Chapter 7

Summary and Conclusions

This chapter summarizes the work performed in this thesis and make remarks of the difficulties found, the lessons learned, and the future work proposed. We also added online content available as an online exhibit is at [Ramirez-Herran and Knill, 2009].

The program implemented was sucessfull in doing the tracking and the reconstruction. In the next sections we recorded the difficulties found in the process of developing this solution and we propose future work. We consider this project has a big potential in become a robust solution to the structure from motion problem for Omnidirectional vision.

7.1 Difficulties of tracking

- There are many parameters which play a role: curvature and gradient thresholds, various patch sizes and search sizes.
- If too many points are tracked, the reconstruction is difficult for linear algebra.
- If too few points are tracked, the tracking is unreliable.

- Tracking needs to be more robust.
- Determining the periods, for which the tracking is reliable. Points disappear and reappear.
- The parameters and thresholds depend very much on the situation, even in synthetic cases. This is why a reliable automatic parameter estimation is important.
- Finding correspondences, is still a challenging problem in practice. The correspondences found by tracking assume constancy in feature appearance and smoothness in camera motion so that the search space for correspondences is much reduced. Tracking is difficult under degenerate conditions, such as occlusions, or abrupt camera motion.

7.2 Difficulties of the reconstruction

The resulting 3D reconstruction is often observed to be extremely sensitive to the error in correspondences, which is often caused by the lack of a robust tracking.

7.3 Future work to do

- Finding the thresholds automatically
- Use interpolation of the tracked particle paths. Use higher derivatives of the paths.
- Stay close to large curvature
- Focus aggressively on relevant parts of the picture
- Do preprocessing of the pictures and store curvature information in a picture, the computations could be sped up.

- Build an interactive system, where parameters can be changed.
- Include tracking of vertical and horizontal lines
- Iteratively find the thresholds that produce a specific number of interesting points in the first frame. And adapt the tresholding to changes in the illumination across frames should improve the tracking, and hence the 3D reconstruction would be more robust.
- Implement the linear algebra functions such as least square in C to boost the performance of the reconstruction pipeline.
- Integrate the SFM reconstruction with the tracking. This way you can run the tracking algorithms and get feedback from the structure and camera motion recovered. This will allow us to reduce the error found in the correspondences and explicitly deal with challenging circumstances such as occlusions, ambiguous scenes and abrupt camera motion.
- Perform auto-calibration of the camera based in guessing a dimension in the first frame maybe by using object recognition and matching with an object database or any other strategy.

7.4 Lessons learned

- In this topic of computer vision it is easy to get side tracked since there are so many things one could do. Already the topic of feature tracking is interesting and challenging by itself. How to choose the important feature points?
- We found that it is important to filter points that show big variations to improve the tracking.

- Use the right software for the right things. Mathematica for example can not deal with movies.
- Experiments are important to determine which parameters and strategies work.
- Too ambitious plans can slow the project down. For example, we initially processed multi-scale versions of the movie, and had to wait too long to get things processed, even so tracking points on a multi-scale array of pictures can be done faster. Another issue is that the complexity of the code became larger, if everything had to be done with arrays of movies.

Bibliography

[A. Torii, 2009] A. Torii, M. Havlena, T. P. (2009). From Google street view to 3d city models. In *Proceedings of the 2009 IEEE International Conference on Computer Vision*.

[Benosman and S.B.Kang, 2001] Benosman, R. and S.B.Kang (2001). *Panoramic vision*. Monographis in Computer Science. New York NY: Springer-Verlag.

[Derpanis, 2004] Derpanis, K. (2004). The Harris corner detector. Retrieved February 2, 2009, from http://www.cvr.yorku.ca/members/gradstudents/kosta/index.html.

[Golub and Van Loan, 1980] Golub, G. H. and Van Loan, C. F. (1980). An analysis of the total least squares problem. *SIAM Journal on Numerical Analysis*, 17(6):883–893.

[Kelley and Jr, 2007] Kelley, S. and Jr, F. H. (2007). *Computer Graphics using OpenGL*. Upper Saddle River NJ: Pearson Prentice Hall.

[Knill and Ramirez-Herran, 2007a] Knill, O. and Ramirez-Herran, O. (2007a). On ullman's theorem in computer vision. Retrieved February 2, 2009, from http://arxiv.org/abs/0708.2442.

[Knill and Ramirez-Herran, 2007b] Knill, O. and Ramirez-Herran, O. (2007b). Space and camera path reconstruction for omni-directional vision. Retrieved February 2, 2009, from http://arxiv.org/abs/0708.2438.

[Knill and Ramirez-Herran, 2007c] Knill, O. and Ramirez-Herran, O. (2007c). A structure from motion inequality. Retrieved February 2, 2009, from http://arxiv.org/abs/0708.2432.

[Lucas and Kanade, 1981] Lucas, B. and Kanade, T. (1981). An iterative image registration technique with an application to stereo vision. pages 674–679. International Joint Conference on Artificial Intelligence.

[Lucas and Kanade, 1991] Lucas, B. and Kanade, T. (1991). Detection and tracking of point features. Carnegie Mellon University Technical Report, CMU-CS-91-132.

[M. Bosse and Teller, 2003] M. Bosse, R. Rikoski, J. L. and Teller, S. (2003). Vanishing points and three-dimensional lines from omni-directional video. *The Visual Computer*, 19 (6):417–430.

[M. Etoh and Hata, 1999] M. Etoh, T. A. and Hata, K. (1999). Estimation of structure and motion parameters for a roaming robot that scans the space. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision. 1*, pages 579–584.

[Nixon and Aguado, 2002] Nixon, M. and Aguado, A. (2002). *Feature Extraction and Image Processing*. London UK: Academic Press, Eslevier.

[Oliensis, 2000] Oliensis, J. (2000). A critique of structure-from-motion algorithms. *Computer Vision and Image Understanding*, 80:172–214.

[P. Tissainayagam, 2004] P. Tissainayagam, D. S. (2004). Assessing the performance of corner detectors for point feature tracking applications. *Image and Vision Computing*, 22:663679.

[Pan et al., 2009] Pan, Q., Reitmayr, G., and Drummond, T. (2009). ProFORMA: Probabilistic Feature-based On-line Rapid Model Acquisition. In *Proc. 20th British Machine Vision Conference (BMVC)*. London.

[Paragios, 2006] Paragios, N. (2006). *Handbook of Mathematical Models in computer Vision*. New York, NY: Springer-Verlag.

[PuigLuis et al., 2008] PuigLuis, GuerreroJosechu, and Sturm, P. (2008). Matching of omindirectional and perspective images using the hybrid fundamental matrix. Retrieved February 2, 2009, from http://perception.inrialpes.fr/Publications/2008/PGS08.

[Ramirez-Herran and Knill, 2009] Ramirez-Herran, J. and Knill, O. (2009). Online exhibit for OMNIVIS. Retrieved February 2, 2009, from http://www.math.harvard.edu/˜knill/3dscan2.

[Rawlinson and Jarvis, 2008] Rawlinson, D. and Jarvis, R. (2008). Ways to tell robots where to go. *IEEE Robotics and Automation*, 15(2).

[S.Birchfeld, 1997] S.Birchfeld (1997). Derivation of kanade-lucas-tomasi tracking equation. Retrieved February 2, 2009, from http://www.ces.clemson.edu/ stb/klt.

[Shi and Tomasi, 1994] Shi, J. and Tomasi, C. (1994). Good features to track. pages 593–600. IEEE Conference on Computer Vision and Pattern Recognition.

[Smith and Brady, 1997] Smith, S. M. and Brady, J. M. (1997). SUSAN—a new approach to low level image processing. *Int. J. Comput. Vision*, 23(1):45–78.

[Sturm, 2002] Sturm, P. (2002). Mixing catadioptric and perspective cameras. In *Workshop on Omnidirectional Vision, Copenhagen, Denmark*, pages 37–44. Retrieved February 2, 2009, from http://perception.inrialpes.fr/Publications/2002/Stu02a.

[Tucker, 2004] Tucker, A. B. (2004). *Computer Science Handbook, Second Edition*. Boca Raton FL: Chapman & Hall/CRC.

[Wei, 1989] Wei, M. S. (1989). The perturbation of consistent least squares problems. *Linear Algebra and its Applications*, 112:231–245.

[Winters et al., 2000] Winters, N., Gaspar, J., Lacey, G., and Santos-Victor, J. (2000). Omni-directional vision for robot navigation. In *Proceedings IEEE Workshop on Omnidirectional Vision*, pages 21–28.

[Y. Yagi and Yachida, 2000] Y. Yagi, K. S. and Yachida, M. (2000). Environmental map generation and egomotion estimation in a dynamic environment for an omnidirectional image sensor. In *ICRA*, pages 3493–3498.

Appendices

Appendix A

Input data for the reconstruction of the synthetic scene

In this appendix we show the pixel coordinates of the corners to be used in the reconstruction.

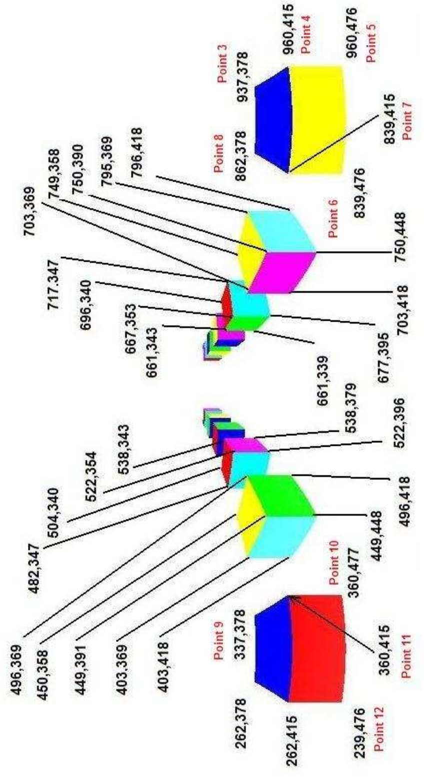


Figure A.1: Corners in the first frame of the Synthetic scene to be reconstructed

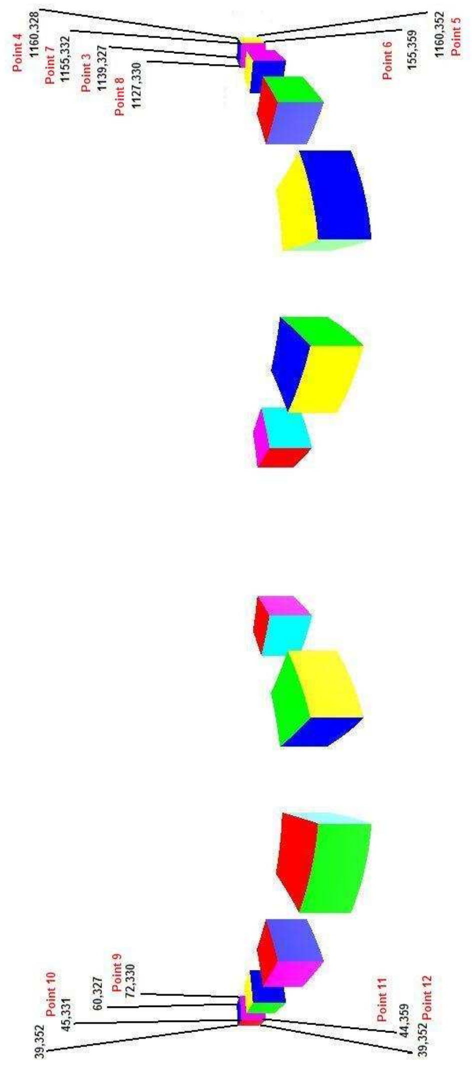


Figure A.2: Corners in the last frame of the Synthetic scene to be reconstructed

Appendix B

Image formats

An image consists of a two-dimensional array of numbers. The color or gray shade displayed for a given picture element (pixel) depends on the number stored in the array for that pixel. The simplest type of image data is black and white. It is a binary image since each pixel is either 0 or 1. The next, more complex type of image data is **gray scale**, where each pixel takes on a value between zero and 256 shades of gray. Since humans can distinguish about 40 shades of gray, 256-shade image is enough. The most complex type of image is **color**. Color images are similar to gray scale except that there are three channels, corresponding to the colors red, green, and blue. Usually, we have one byte = 8 bits for each of the red, green and blue parts so that a color pixel is encoded with 24 bits.

The Netpbm Formats.
Netpbm is a toolkit for manipulation of graphic images, including conversion of images between a variety of different formats. All the programs work with a set of graphic formats called the "netpbm" formats. Specifically, these formats are pbm, pgm, ppm, and pam. The first three are sometimes known generically as "pnm". These are all raster formats, i.e. they describe an image as a matrix of rows and columns of pixels.

The fundamental mission of Netpbm is to make the writing image manipulation programs easy to use. In the PBM format, the pixels are black and white. In the PGM format, pixels are shades of gray. In the PPM format, the pixels are in full color. The PAM is more sophisticated and can represent matrices of general data including but not limited to black and white, grayscale, and color images.

Each of the Netpbm formats has two optional signatures or file descriptors: one corresponding to ASCII version and the second to the bynary version. The ASCII based formats allow for human-readability and easy transport to other platforms, while binary formats are more efficient both saving space in the file, as wel as being easier to parse due to the lack of whitesapace. Specifically PBM file descriptors are P1 and P4. PBM uses 1 bit per pixel. PGM file descriptors are P2 and P5. PGM uses 8 bits per pixel. And PPM file descriptors are P3 and P6. PPM uses 24 bits per pixel: 8 for red, 8 for green, 8 for blue.

The ppm Format.
In this thesis we use the pnm color format named PPM. It is a format that make writing programs for color image manipulation easy and can be converted to any other format easy. All the pnm formats are designed to be as simple as possible. Each starts with a header and the bitmap data follows immediatelly after. The header is always written in ASCII, and the data items are separated by white space. The data can be writen in either ASCII or binary form.

The PPM Header.
Consist of the following entries, each separated by white space: MagicValue or file descriptor: P3 for ASCII version, P6 for binary Image Width is the widht of image in pixels (ASCII decimal value) Image Height is the height of image in pixels (ASCII decimal value)

MaxGrey is the Maximum color value,i.e, its ASCII decimal value

The PPM Image Data.

The following is an example of the ppm file format P3

3 2

255

255 0 0 0 255 0 0 0 255

255 255 0 255 255 255 0 0 0

And the image expanded is

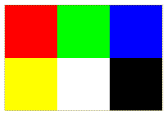


Figure B.1: Example of a P3 (ASCII) PPM format image.

The P6 binary format of the same image will store easch color component of each pixel with one byte. The pixel values are stored as plain bytes, instead of ASCII decimal. The PPM Rawbits is three bytes per pixel: 8 bits for red, 8 bits for greeen, 8 bits for blue. Note that this raw format can only be use with values less than or equal to 255. The file will be smaller but the color information will not be readable by humans.

The PPM is not compressed and thus requires more space and bandwidth than a compressed format would require. For example, a particular 192x128 PNG image has a file size of 187 bytes. When converted to a 192x128 PPM image, the file size is 73848 bytes. The PPM format is an intermediate format used for image work before converting to a more efficient format, for example the PNG format, without loss of information in the intermediate step.

Appendix C

Source Code

```
/*----------------------------------------------------------------------------*/
/*  vectorfield.c                                                             */
/*----------------------------------------------------------------------------*/

/*\
 *  This file is part of the Omnivis project, Harvard University
 *  Copyright (C) December 2009  Oliver Knill and Jose Ramirez Herran
 *  http://www.math.harvard.edu/~knill/3dscan2           (project page)
 *
 *  Usage:  gcc -o vectorfield -lm vectorfield.c;
 *          ./vectorfield ‘ls *.pnm‘
 *
 *  In a first run, it will produce auxiliary files in a directory tmp
 *  like curvature, gradient and smoothed pictures
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
\*/

#include <stdio.h>                          // reading in standard libraries
#include <stdlib.h>                         // this compiles in any unix
#include <string.h>                         // environment with gcc
#include <math.h>
```

```
#define PI 3.14159265358979323846           // pi
#define FILE_INDEX(x) "",(x)                // to generate file names

#define MP 10000                            // maximal number of points to track and print
#define NF 300                              // maximal number of input files
#define S  4                                // how many times is the picture smoothed first
#define PS 4                                // patch inflation size for curvature
#define PS1 2                               // patch inflation size for correspondence
#define RR 5                                // length of slope line which is drawn if drawn at all
#define M 10                                // max search distance for best nearby pattern
#define N 5                                 // size of pattern patch to compare
#define K 3                                 // search grid dimension, look at each K'th pixel
#define KK 1                                // grid size to average derivatives over
#define bb 20                               // boundary width to search
#define merge 3                             // radius within which we merge points
double  threshold =  6000;                  // gradient      New: is now be determined
#define ethreshold 100                      // threshold for edges    New: will now be determined
#define scaledthresh 170                    // new threshold scaled to 255,  70 for florida, 150 for cubes
#define stroke  100                         // how long to draw red stream lines for Hamiltonian
double alpha=PI/2+0*PI/2;                   // PI/2 for Hamiltonian ,  PI for gradient
int  preprocessed;                          // =1 if pictures are in tmp, otherwise 0, meaning we have to pre process
int  smootheddone;                          // =1 if smoothed pictures are in tmp, otherwise 0

int  x_match[MP][NF];                       // x-coordinate of a matched point, [point number,camera number]
int  y_match[MP][NF];                       // y-coordinate of a matched point, [point number,camera number]
int  countpoints[NF];                       // number of points which qualify

FILE *out[NF];                              // output file
FILE *smoothout[NF];                        // output file for smooth pictures
FILE *outcurvature[NF];                     // output file for curvature pictures
FILE *outgradient[NF];                      // output file for gradient pictures
FILE *in[NF];                               // input file
FILE *insmoothed[NF];                       // smoothed input file
FILE *incurvature[NF];                      // curvature input file
FILE *ingradient[NF];                       // gradient input file

char *infilename[NF];                       // input file name
char *outfilename[NF];                      // output file name
char *smoothfilename[NF];                   // output file name for smooth pictures
char *curvaturefilename[NF];                // output file name for curvature pictures
char *gradientfilename[NF];                 // output file name for gradient pictures
char *outputarray[NF];                      // pointers for output
char *outcurvaturearray[NF];                // pointers for curvature output
char *outgradientarray[NF];                 // pointers for gradient
char *inputarray[NF];                       // pointer for input
char *insmoothedarray[NF];                  // pointers for smoothed input
char *incurvaturearray[NF];                 // pointers for curvature input
char *ingradientarray[NF];                  // pointers for gradient input

int  x_size,y_size,c_size;                  // width, height and number of colors
int  midframe;                              // equal to 1 for NF=3,  equal to 1 for NF=4

long gaussian[5][5];                        // Gaussian kernel
long laplace[5][5];                         // Laplace operator derivative
long hessianxx[5][5];                       // Hessian xx
long hessianxy[5][5];                       // Hessian xy
```

```
long hessianyy[5][5];                       // Hessian yy
long sobelx[5][5];                          // Sobel partial x derivative
long sobely[5][5];                          // Sobel partial y derivative
long rpatch[5][5];                          // Patch for smoothing
long gpatch[5][5];                          // Patch for smoothing
long bpatch[5][5];                          // Patch for smoothing
long gaussiansum;                           // sum of Gaussian kernel entries

char *ptr;                                  // general pointer for output picture

long    rx,ry,gx,gy,bx,by;                  // red green blue value of two pictures
long    rthresh,rthreshx,rthreshy;
long    gthresh,gthreshx,gthreshy;
long    bthresh,bthreshx,bthreshy;          // threshold functions
double  thresh,edge,cross;                  // average over colors
double  maxthresh, maxedge,maxcross;        // maximal values
int     imaxthresh, imaxedge,imaxcross;     // maximal values
int     maxrthresh[NF];
int     maxgthresh[NF];
int     maxbthresh[NF];                     // maximal gradient values
int     maxredge[NF];
int     maxgedge[NF];
int     maxbedge[NF];                       // maximal curvature values
int     redge,gedge,bedge;                  // edge values
long    rcross,gcross,bcross;               // cross values
double  rang,gang,bang;                     // red green blue angle
double  rcos,gcos,bcos;                     // cos(angles) of gradients
double  rsin,gsin,bsin;                     // sin(angles) of gradients

/*---------------------------------------------------------------------------*/
/*  Define first and second derivatives of the red,green and blue functions  */
/*---------------------------------------------------------------------------*/

void definematrices() {
  gaussian[0][0]=1; gaussian[0][1]=3; gaussian[0][2]=4; gaussian[0][3]=3; gaussian[0][4]=1;
  gaussian[1][0]=3; gaussian[1][1]=7; gaussian[1][2]=10;gaussian[1][3]=7; gaussian[1][4]=3;
  gaussian[2][0]=4; gaussian[2][1]=10;gaussian[2][2]=14;gaussian[2][3]=10;gaussian[2][4]=4;
  gaussian[3][0]=3; gaussian[3][1]=7; gaussian[3][2]=10;gaussian[3][3]=7; gaussian[3][4]=3;
  gaussian[4][0]=1; gaussian[4][1]=3; gaussian[4][2]=4; gaussian[4][3]=3; gaussian[4][4]=1;
  gaussiansum=126;

  sobely[0][0]=0; sobely[0][1]=0;  sobely[0][2]=0; sobely[0][3]=0; sobely[0][4]=0;
  sobely[1][0]=0; sobely[1][1]=-1; sobely[1][2]=0; sobely[1][3]=1; sobely[1][4]=0;
  sobely[2][0]=0; sobely[2][1]=-2; sobely[2][2]=0; sobely[2][3]=2; sobely[2][4]=0;
  sobely[3][0]=0; sobely[3][1]=-1; sobely[3][2]=0; sobely[3][3]=1; sobely[3][4]=0;
  sobely[4][0]=0; sobely[4][1]=0;  sobely[4][2]=0; sobely[4][3]=0; sobely[4][4]=0;

  sobelx[0][0]=0; sobelx[0][1]=0;  sobelx[0][2]=0;  sobelx[0][3]=0;  sobelx[0][4]=0;
  sobelx[1][0]=0; sobelx[1][1]=1;  sobelx[1][2]=2;  sobelx[1][3]=1;  sobelx[1][4]=0;
  sobelx[2][0]=0; sobelx[2][1]=0;  sobelx[2][2]=0;  sobelx[2][3]=0;  sobelx[2][4]=0;
  sobelx[3][0]=0; sobelx[3][1]=-1; sobelx[3][2]=-2; sobelx[3][3]=-1; sobelx[3][4]=0;
  sobelx[4][0]=0; sobelx[4][1]=0;  sobelx[4][2]=0;  sobelx[4][3]=0;  sobelx[4][4]=0;

  laplace[0][0]=0; laplace[0][1]=0;  laplace[0][2]=0;  laplace[0][3]=0;  laplace[0][4]=0;
  laplace[1][0]=0; laplace[1][1]=0;  laplace[1][2]=1;  laplace[1][3]=0;  laplace[1][4]=0;
  laplace[2][0]=0; laplace[2][1]=1;  laplace[2][2]=-4; laplace[2][3]=1;  laplace[2][4]=0;
  laplace[3][0]=0; laplace[3][1]=0;  laplace[3][2]=1;  laplace[3][3]=0;  laplace[3][4]=0;
  laplace[4][0]=0; laplace[4][1]=0;  laplace[4][2]=0;  laplace[4][3]=0;  laplace[4][4]=0;

  hessianyy[0][0]= 0; hessianyy[0][1]= 0;  hessianyy[0][2]= 0;  hessianyy[0][3]=0;  hessianyy[0][4]=0;
  hessianyy[1][0]= 0; hessianyy[1][1]= 1;  hessianyy[1][2]=-2;  hessianyy[1][3]=1;  hessianyy[1][4]=0;
  hessianyy[2][0]= 0; hessianyy[2][1]= 2;  hessianyy[2][2]=-4;  hessianyy[2][3]=2;  hessianyy[2][4]=0;
  hessianyy[3][0]= 0; hessianyy[3][1]= 1;  hessianyy[3][2]=-2;  hessianyy[3][3]=1;  hessianyy[3][4]=0;
  hessianyy[4][0]= 0; hessianyy[4][1]= 0;  hessianyy[4][2]= 0;  hessianyy[4][3]=0;  hessianyy[4][4]=0;

  hessianxx[0][0]=0; hessianxx[0][1]=0;  hessianxx[0][2]=0;  hessianxx[0][3]=0;  hessianxx[0][4]=0;
  hessianxx[1][0]=0; hessianxx[1][1]=1;  hessianxx[1][2]=2;  hessianxx[1][3]=1;  hessianxx[1][4]=0;
  hessianxx[2][0]=0; hessianxx[2][1]=-2; hessianxx[2][2]=-4; hessianxx[2][3]=-2; hessianxx[2][4]=0;
  hessianxx[3][0]=0; hessianxx[3][1]=1;  hessianxx[3][2]=2;  hessianxx[3][3]=1;  hessianxx[3][4]=0;
  hessianxx[4][0]=0; hessianxx[4][1]=0;  hessianxx[4][2]=0;  hessianxx[4][3]=0;  hessianxx[4][4]=0;

  hessianxy[0][0]=0; hessianxy[0][1]= 0; hessianxy[0][2]=0;  hessianxy[0][3]=0;  hessianxy[0][4]=0;
  hessianxy[1][0]=0; hessianxy[1][1]=-4; hessianxy[1][2]=0;  hessianxy[1][3]=4;  hessianxy[1][4]=0;
  hessianxy[2][0]=0; hessianxy[2][1]= 0; hessianxy[2][2]=0;  hessianxy[2][3]=0;  hessianxy[2][4]=0;
  hessianxy[3][0]=0; hessianxy[3][1]= 4; hessianxy[3][2]=0;  hessianxy[3][3]=-4; hessianxy[3][4]=0;
  hessianxy[4][0]=0; hessianxy[4][1]= 0; hessianxy[4][2]=0;  hessianxy[4][3]= 0; hessianxy[4][4]=0;
}

/*---------------------------------------------------------------------------*/
/*  Draw routines of points, lines and arrows                                */
/*---------------------------------------------------------------------------*/

void draw_point(int a,int xp,int yp,int r,int g,int b){
  ptr=outputarray[a]+3*((-yp)*x_size+xp); *ptr++=r; *ptr++=g; *ptr=b;
}

void draw_line(int a,int xa,int ya, int xb,int yb,int r,int g,int b){
   int l; int L;
   int lx,ly;
   if (xa<0) { xa=0; } if (xa> x_size) { xa= x_size; } if (xb< 1) { xb=0; } if (xb> x_size) { xb= x_size; }
   if (ya>0) { ya=0; } if (ya<-y_size) { ya=-y_size; } if (yb>-1) { yb=0; } if (yb<-y_size) { yb=-y_size; }
   L=abs(xb-xa)+abs(yb-ya);
   if (L==0) { draw_point(a,xa,ya,r,g,b); } else {
    for (l=0; l<=L; l++) {
      lx=floor(1.0*l*(xb-xa)/L);
      ly=floor(1.0*l*(yb-ya)/L);
      draw_point(a,xa+lx,ya+ly,r,g,b);
    }
   }
}

void draw_arrow(int a,int xa,int ya, int xb,int yb,int r,int g,int b){
   int vx,vy,wx,wy,xc,yc,xd,yd,xe,ye;
     draw_line(a,xa,ya,xb,yb,r,g,b);                    // base of arrow
     vx=xb-xa; vy=yb-ya;  wx=-vy; wy=vx;
     xc=floor((xa+1*xb)/2); yc=floor((ya+1*yb)/2);
     xd=floor(xc+3*wx/4);   yd=floor(yc+3*wy/4);
     xe=floor(xc-3*wx/4);   ye=floor(yc-3*wy/4);
     draw_line(a,xe,ye,xd,yd,r,g,b);                    // head of arrow
     draw_line(a,xb,yb,xd,yd,r,g,b);                    // is a triangle
     draw_line(a,xb,yb,xe,ye,r,g,b);                    //
}
```

```
long  chartoint(char cen) { long a=cen; if (a<0) a+=256; return(a); }  //  convert char to integer
int xwrap(int x){int xx=(x+x_size) % x_size; return(xx); }             //  wrapping functions
int ywrap(int y){int yy=(y+y_size) % y_size; return(yy); }             //  wrapping function
int huetored(float h){float r=0.0; int u; float f,q; u=floor(5.999*h); f=5.999*h-u; q=1-f;
      if(u==0 || u==5) {r=1.0;} if(u==1){r=q;} if(u==4){r=f;} return(floor(255.0*r));}
int huetogreen(float h){float r=0.0; int u; float f,q; u=floor(5.999*h); f=5.999*h-u; q=1-f;
      if(u==1 || u==2) {r=1.0;} if(u==3){r=q;} if(u==0){r=f;} return(floor(255.0*r));}
int huetoblue(float h){float r=0.0; int u; float f,q; u=floor(5.999*h); f=5.999*h-u; q=1-f;
      if(u==3 || u==4) {r=1.0;} if(u==5){r=q;} if(u==2){r=f;} return(floor(255.0*r));}
double angle(int xx, int yy) {     // compute arg(x+i y), in C atan takes values in (-pi/2,pi/2)
double ang;
  if(yy==0) { if (xx>0) {ang=0.0;  } else {ang=PI;}     } else {
  if(xx==0) { if (yy>0) {ang=PI/2; } else {ang=3*PI/2;} } else {
  if(xx>0)  { ang=atan(1.0*yy/xx);}  else {ang=atan(1.0*yy/xx) + PI;}}}
  ang+=alpha;
return(ang); }

/*---------------------------------------------------------------------------*/
/*  Gradient and curvature computations  in color space                      */
/*---------------------------------------------------------------------------*/

void curvature1(int a, int iii, int jjj){
  long    rxx,gxx,bxx;                      // store variables
  long    ryy,gyy,byy;                      // store variables
  long    rxy,gxy,bxy;                      // store variables
  int     u,v;                              // summing over patch
  int     ii,jj;                            // indices for patch
  char    *inptr;                           // for gradient computation
  long    rthresh1,gthresh1,bthresh1;


    for (u=-2; u<=2; u++) {
      for (v=-2; v<=2; v++) {
        ii=xwrap(iii+PS*u);jj=-ywrap(-jjj+PS*v);
        inptr=inputarray[a]+3*((-jj)*x_size+ii);
        rpatch[u+2][v+2]=chartoint(*inptr++);gpatch[u+2][v+2]=chartoint(*inptr++);bpatch[u+2][v+2]=chartoint(*inptr);
      }
      }

      // first derivatives
      rx=0; ry=0; gx=0; gy=0; bx=0; by=0;
      for (u=1; u<=3; u+=2) {
      for (v=1; v<=3; v+=2) {
        rx += sobelx[u][v]*rpatch[u][v];   gx+=sobelx[u][v]*gpatch[u][v];   bx+=sobelx[u][v]*bpatch[u][v];
        ry += sobely[u][v]*rpatch[u][v];   gy+=sobely[u][v]*gpatch[u][v];   by+=sobely[u][v]*bpatch[u][v];
      }
      }

      // second derivatives
      rxx=0; ryy=0; rxy=0; gxx=0; gyy=0; gxy=0; bxx=0; byy=0; bxy=0;
      for (u=1; u<=3; u+=1) {
      for (v=1; v<=3; v+=1) {
        rxx+=hessianxx[u][v]*rpatch[u][v];  gxx+=hessianxx[u][v]*gpatch[u][v]; bxx+=hessianxx[u][v]*bpatch[u][v];
        ryy+=hessianyy[u][v]*rpatch[u][v];  gyy+=hessianyy[u][v]*gpatch[u][v]; byy+=hessianyy[u][v]*bpatch[u][v];
        rxy+=hessianxy[u][v]*rpatch[u][v];  gxy+=hessianxy[u][v]*gpatch[u][v]; bxy+=hessianxy[u][v]*bpatch[u][v];
      }
      }
      // unsigned curvature, where |grad(f)|^3 is replaced by |grad(f)|^2 to get larger integers
      // it just works so much better than the Kitchen-Rosenfeld curvature
      rthresh=(rx*rx+ry*ry); gthresh=(gx*gx+gy*gy); bthresh=(bx*bx+by*by);
      if (rthresh==0) {redge=0;} else {redge=floor(abs((rxx*ry*ry-2*rxy*rx*ry+ryy*rx*rx)*1.0)/rthresh);}
      if (gthresh==0) {gedge=0;} else {gedge=floor(abs((gxx*gy*gy-2*gxy*gx*gy+gyy*gx*gx)*1.0)/gthresh);}
      if (bthresh==0) {bedge=0;} else {bedge=floor(abs((bxx*by*by-2*bxy*bx*by+byy*bx*bx)*1.0)/bthresh);}
}

void curvature(int a, int iii, int jjj){
  int     u,v;
  int     KKK = (2*KK+1)*(2*KK+1);
  int     rrthresh,ggthresh,bbthresh,rredge,ggedge,bbedge;
  long    rrx,rry,ggx,ggy,bbx,bby;

  rrthresh=0;ggthresh=0;bbthresh=0;rredge=0;ggedge=0;bbedge=0;
  rrx=0; ggx=0; bbx=0; rry=0; ggy=0; bby=0;

  for (u=-KK; u<=KK; u++) {
  for (v=-KK; v<=KK; v++) {
      curvature1(a,iii+u,jjj+v);
      rrthresh+=rthresh; ggthresh+=gthresh; bbthresh+=bthresh;
      rredge  +=redge;   ggedge  +=gedge;   bbedge  +=bedge;
      rrx     +=rx;      ggx     +=gx;      bbx     +=bx;
      rry     +=ry;      ggy     +=gy;      bby     +=by;
  }
  }

  rthresh=floor(rrthresh/KKK); gthresh=floor(ggthresh/KKK); bthresh=floor(bbthresh/KKK);
  redge=floor(rredge/KKK);     gedge  =floor(ggedge/KKK);   bedge=floor(bbedge/KKK);
  rx=rrx/KKK; gx=ggx/KKK;      bx=bbx/KKK; ry=rry/KKK;      gy=ggy/KKK; by=bby/KKK;

  rang = angle(rx,ry);  if( rx==0 && ry==0) { rcos=0.0; rsin=0.0;} else { rcos = cos(rang); rsin = sin(rang); }
  gang = angle(rx,ry);  if( gx==0 && gy==0) { gcos=0.0; gsin=0.0;} else { gcos = cos(gang); gsin = sin(gang); }
  bang = angle(rx,ry);  if( bx==0 && by==0) { bcos=0.0; bsin=0.0;} else { bcos = cos(bang); bsin = sin(bang); }
}

/*---------------------------------------------------------------------------*/
/*  Side track fancy pictures, where red,green and blue pens along levels    */
/*---------------------------------------------------------------------------*/

void drawhamiltonian(a){
  int     i,j,k;                          // indices
  int     ii,jj,ii0,jj0,ii1,jj1;          // pixel positions
  double  rii,gii,bii,rjj,gjj,bjj;        // real positions

    for(j = -bb; ( (j > (-y_size+bb)) ); j-=K){
    for(i =  bb; ( (i < ( x_size-bb)) ); i+=K){

      curvature(a,i,j);
      rii=1.0*i; rjj=1.0*j; ii=i; jj=j; k=0;  // draw red flow lines
      while  ((rthresh>threshold) && (jj > (-y_size+bb)) && (jj<-bb) && (ii < (x_size-bb)) && (ii>bb) && k<stroke) {
        curvature(a,ii,jj); k++;
        if (rthresh>threshold) {
          rii+=rcos; rjj+=rsin; ii0=ii; jj0=jj; ii=(int) rii; jj=(int) rjj;
          ii1= (int) RR*rcos; jj1=(int) RR*rsin;
          draw_line(a,ii0,jj0,ii,jj,255,20,20);
```

```
        }
      }

      curvature(a,i,j);
      gii=1.0*i; gjj=1.0*j; ii=i; jj=j; k=0;  // draw green flow lines
      while  ((gthresh>threshold) && (jj > (-y_size+bb)) && (jj<-bb) && (ii < (x_size-bb)) && (ii>bb) && k<stroke) {
        curvature(a,ii,jj); k++;
        if (gthresh>threshold) {
          gii+=gcos; gjj+=gsin; ii0=ii; jj0=jj; ii=(int) gii; jj=(int) gjj;
          ii1= (int) RR*gcos; jj1=(int) RR*gsin;
          draw_line(a,ii0,jj0,ii,jj,20,255,20);
        }
      }

      curvature(a,i,j);
      bii=1.0*i; bjj=1.0*j; ii=i; jj=j; k=0;  // draw blue flow lines
      while  ((bthresh>threshold) && (jj > (-y_size+bb)) && (jj<-bb) && (ii < (x_size-bb)) && (ii>bb) && k<stroke) {
        curvature(a,ii,jj); k++;
        if (bthresh>threshold) {
          bii+=bcos; bjj+=bsin; ii0=ii; jj0=jj; ii=(int) bii; jj=(int) bjj;
          ii1= (int) RR*bcos; jj1=(int) RR*bsin;
          draw_line(a,ii0,jj0,ii,jj,20,20,255);
        }
      }

   }                                          // end of i loop
   }                                          // end of j loop
}

/*---------------------------------------------------------------------------*/
/*  To gauge the curvature spectrum, we make a dry run                       */
/*---------------------------------------------------------------------------*/


void find_parameters(int a){                  // we have to know the maximal curvatures before
int     i,j;                                  // we can write the curvature files
   fprintf(stderr,"Find parameters: ");

    maxthresh=0;     maxedge=0;
    maxrthresh[a]=1; maxgthresh[a]=1; maxbthresh[a]=1;
    maxredge[a]=1;   maxgedge[a]=1;   maxbedge[a]=1;

    for (j = -bb; ( (j > (-y_size+bb)) ); j-=K){                    // go through the entire picture
    for (i =  bb; ( (i < ( x_size-bb)) ); i+=K){

      curvature1(a,i,j);

      if (rthresh>maxrthresh[a]) { maxrthresh[a]=rthresh; }         // maximal red green blue gradients
      if (gthresh>maxgthresh[a]) { maxgthresh[a]=gthresh; }
      if (bthresh>maxbthresh[a]) { maxbthresh[a]=bthresh; }

      if (redge>maxredge[a]) { maxredge[a]=redge; }                 // maximal red green blue curvatures
      if (gedge>maxgedge[a]) { maxgedge[a]=gedge; }
      if (bedge>maxbedge[a]) { maxbedge[a]=bedge; }
    }
    }
   fprintf(stderr,"Curvatures %i %i %i \n",maxredge[a],maxgedge[a],maxbedge[a]);
}

/*----------------------------------------------------------------------------*/
/*  At the beginning, we have to decide which points to take                  */
/*----------------------------------------------------------------------------*/

void initial_vectorfield(int a){
  int     i,j,k,l,m,p,q,u,v;                // indices
  int     ii,jj,kk,ll;                      // points in picture
  int     i0,j0,i2,j2;                      // points in previous and next picture
  long    laplacer,laplaceg,laplaceb;       // Laplace values
  double  diff[NF], mindiff[NF];            // best differences
  long    r0,r1,r2,g0,g1,g2,b0,b1,b2;       // red green blue value of two pictures
  int     minp[NF], minq[NF];               // the best translation vector
  int     mx,my;                            // for matching
  int     dist;                             // length of line in taximetric
  int     cp;                               // point index

    fprintf(stderr,"Compute initial vector field. ");

    cp=0;
    for(j = -bb; ( (j > (-y_size+bb)) ); j-=K){                // go through the entire picture
    for(i =  bb; ( (i < ( x_size-bb)) ); i+=K){

      ptr=incurvaturearray[a]+3*((-j)*x_size+i);
      redge=chartoint(*ptr++); gedge=chartoint(*ptr++); bedge=chartoint(*ptr);

      if ( redge>scaledthresh  || gedge>scaledthresh || bedge>scaledthresh )   {    //  interesting pts

          mindiff[a-1]=10000.0; mindiff[a+1]=10000.0;
          mx=xwrap(i); my=-ywrap(-j);
          if( cp<MP) {
             x_match[cp][a]=mx; y_match[cp][a]=my; cp++; countpoints[a]=cp;
          }
          minp[a-1]=mx; minq[a-1]=my;
          minp[a+1]=mx; minq[a+1]=my;

          for (q =  M; q >= -M; q--) {                       // begin passing over displacement vector
          for (p = -M; p <=  M; p++) {

          mx=xwrap(i+q); my=-ywrap(-j-p); diff[a-1]=0.0;  diff[a+1]=0.0;

          for (v =  N; v >= -N; v--) {                       // begin summing over patch
          for (u = -N; u <=  N; u++) {
              ii=xwrap(i+PS1*u);  jj=-ywrap(-j-PS1*v);
              kk=xwrap(i+PS1*u+q);ll=-ywrap(-j-PS1*v-p);

              ptr=inputarray[a]+3*((-jj)*x_size+ii);
              r1=chartoint(*ptr++); g1=chartoint(*ptr++); b1=chartoint(*ptr);

              ptr=inputarray[a-1]+3*((-ll)*x_size+kk);
              r2=chartoint(*ptr++); g2=chartoint(*ptr++); b2=chartoint(*ptr);
              diff[a-1]+= abs(r1-r2)+abs(g1-g2)+abs(b1-b2);

              ptr=inputarray[a+1]+3*((-ll)*x_size+kk);
              r2=chartoint(*ptr++); g2=chartoint(*ptr++); b2=chartoint(*ptr);
```

```
              diff[a+1]+= abs(r1-r2)+abs(g1-g2)+abs(b1-b2);
          }
          }                                                   // end summing over patch

          if (diff[a-1]<mindiff[a-1]) { minp[a-1]=mx; minq[a-1]=my; mindiff[a-1]=diff[a-1]; }
          if (diff[a+1]<mindiff[a+1]) { minp[a+1]=mx; minq[a+1]=my; mindiff[a+1]=diff[a+1]; }

          }
          }                                                   // end passing over displacement vector

       x_match[cp][a+1]=minp[a+1]; y_match[cp][a+1]=minq[a+1];
       dist= abs(minp[a-1]-minp[a+1]) +  abs(minq[a-1]-minq[a+1]);
       if (mindiff[a-1]>1 && mindiff[a+1]>1 && dist<x_size/2 && dist<y_size/2) {
         draw_line(1,minp[a-1],minq[a-1],minp[a+1],minq[a+1],255,  0,  0);
       }

     }

   }
   }

   fprintf(stderr,"Number of points: %i \n", countpoints[a]);
   fprintf(stderr,"Tracking: ");
}                                           //  end initiate_vectorfield

/*----------------------------------------------------------------------------*/
/*  Now, we evolve the chosen points along the motion field                   */
/*----------------------------------------------------------------------------*/

void vectorfield(int a){
  int     i,j,k,l,m,p,q,u,v;                // indices
  int     ii,jj,kk,ll;                      // points in picture
  int     i0,j0,i2,j2;                      // points in previous and next picture
  long    laplacer,laplaceg,laplaceb;       // Laplace values
  double  thresh,edge;                      // average over colors
  double  diff[NF], mindiff[NF];            // best differences
  double  localthresh;                      // old threshold in neighborhood
  double  localedge;                        // old edge in neighborhood
  long    r0,r1,r2,g0,g1,g2,b0,b1,b2;       // red green blue value of two pictures
  int     minp[NF], minq[NF];               // the best translations
  int     mx,my,mx0,my0;                    // for matching
  int     dist;                             // length of line in taximetric
  int     cp;

  fprintf(stderr,".");
  for (cp=0; cp<countpoints[a]; cp++) {
    i = x_match[cp][a-1]; j=y_match[cp][a-1];

    ptr=incurvaturearray[a]+3*((-j)*x_size+i);
    redge=chartoint(*ptr++); gedge=chartoint(*ptr++); bedge=chartoint(*ptr);
    ptr=ingradientarray[a]+3*((-j)*x_size+i);
    rthresh=chartoint(*ptr++); gthresh=chartoint(*ptr++); bthresh=chartoint(*ptr);

    mx=xwrap(i); my=-ywrap(-j);  mx0=mx; my0=my;
    minp[a]=mx; minq[a]=my; mindiff[a]=10000.0;

          for (q =  M; q >= -M; q--) {                       // begin passing over displacement vector
          for (p = -M; p <=  M; p++) {

          mx=xwrap(i+q); my=-ywrap(-j-p); diff[a]=0.0;

          for (v =  N; v >= -N; v--) {                       // begin summing over patch
          for (u = -N; u <=  N; u++) {
              ii=xwrap(i+PS1*u);  jj=-ywrap(-j-PS1*v);
              kk=xwrap(i+PS1*u+q);ll=-ywrap(-j-PS1*v-p);
              ptr=inputarray[a-1]+3*((-jj)*x_size+ii);
              r1=chartoint(*ptr++); g1=chartoint(*ptr++); b1=chartoint(*ptr);
              ptr=inputarray[a]+3*((-ll)*x_size+kk);
              r2=chartoint(*ptr++); g2=chartoint(*ptr++); b2=chartoint(*ptr);
              diff[a]+= abs(r1-r2)+abs(g1-g2)+abs(b1-b2);
          }
          }                                                   // end summing over patch

          if (diff[a]<mindiff[a]) { minp[a]=mx; minq[a]=my; mindiff[a]=diff[a]; }

          }
          }                                                   // end passing over displacement vector

       x_match[cp][a]=minp[a]; y_match[cp][a]=minq[a];
       dist= abs(mx0-minp[a]) + abs(my0-minq[a]);

       if (mindiff[a]>1 && dist<x_size/2  && dist<y_size/2) {
          draw_line(a,mx0,my0,minp[a],minq[a],255,  0,  0);
          draw_line(1,mx0,my0,minp[a],minq[a],255,  0,  0);
       }
    }
}

/*---------------------------------------------------------------------------*/
/*  Produce sequences of file names for various external files               */
/*---------------------------------------------------------------------------*/


char *make_filename_a(int a, char *ext){            // for output pictures
  char *basename = "a";
  char *filename;
  filename = (char *) malloc((strlen(basename)+strlen(ext)+5)*sizeof(char));
  sprintf(filename,"%s%s%i.%s",basename,FILE_INDEX(a),ext);
  return(filename);
}

char *make_filename_s(int a, char *ext){          // for smooth pictures
  char *basename = "tmp/s";
  char *filename;
  filename = (char *) malloc((strlen(basename)+strlen(ext)+9)*sizeof(char));
  sprintf(filename,"%s%s%i.%s",basename,FILE_INDEX(a),ext);
  return(filename);
}

char *make_filename_c(int a, char *ext){          // for curvature pictures
  char *basename = "tmp/c";
  char *filename;
  filename = (char *) malloc((strlen(basename)+strlen(ext)+9)*sizeof(char));
  sprintf(filename,"%s%s%i.%s",basename,FILE_INDEX(a),ext);
```

```
  return(filename);
}

char *make_filename_g(int a, char *ext){          // for curvature pictures
  char *basename = "tmp/g";
  char *filename;
  filename = (char *) malloc((strlen(basename)+strlen(ext)+9)*sizeof(char));
  sprintf(filename,"%s%s%i.%s",basename,FILE_INDEX(a),ext);
  return(filename);
}

void write_output(int a){                         // write the pnm outputfile
  int   i,j,k;
  char *ptr0;
  out[a]=fopen(outfilename[a],"wb");
  fprintf(out[a],"P6\n%i %i\n%i\n", x_size, y_size, c_size);
    for(j = 0; j > -y_size; j--){
    for(i = 0; i <  x_size; i++){
      ptr0  = outputarray[a] + 3*((-j)*x_size+i);
      fputc(*ptr0++,out[a]); fputc(*ptr0++,out[a]); fputc(*ptr0,out[a]);
    }
    }
  fclose(out[a]);
}

/*---------------------------------------------------------------------------*/
/*  Write external files which are ppm pictures                              */
/*---------------------------------------------------------------------------*/

void write_smoothed(int a){                       // write the pnm smoothed file
  int   i,j,k;
  char *ptr0;
  smoothout[a]=fopen(smoothfilename[a],"wb");
  fprintf(smoothout[a],"P6\n%i %i\n%i\n", x_size, y_size, c_size);
    for(j = 0; j > -y_size; j--){
    for(i = 0; i <  x_size; i++){
      ptr0  = outputarray[a] + 3*((-j)*x_size+i);
      fputc(*ptr0++,smoothout[a]);
      fputc(*ptr0++,smoothout[a]);
      fputc(*ptr0,smoothout[a]);
    }
    }
  fclose(smoothout[a]);
}

void write_curvature(int a){                      // write the curvature file
  int   i,j,k;
  char *ptr0;
  outcurvature[a]=fopen(curvaturefilename[a],"wb");
  fprintf(outcurvature[a],"P6\n%i %i\n%i\n", x_size, y_size, c_size);
    for(j = 0; j > -y_size; j--){
    for(i = 0; i <  x_size; i++){
      ptr0  = outcurvaturearray[a] + 3*((-j)*x_size+i);
      fputc(*ptr0++,outcurvature[a]);
      fputc(*ptr0++,outcurvature[a]);
      fputc(*ptr0,outcurvature[a]);
    }
    }
  fclose(outcurvature[a]);
}



void write_gradient(int a){                        // write the gradients
  int   i,j,k;
  char *ptr0;
  outgradient[a]=fopen(gradientfilename[a],"wb");
  fprintf(outgradient[a],"P6\n%i %i\n%i\n", x_size, y_size, c_size);
    for(j = 0; j > -y_size; j--){
    for(i = 0; i <  x_size; i++){
      ptr0  = outgradientarray[a] + 3*((-j)*x_size+i);
      fputc(*ptr0++,outgradient[a]);
      fputc(*ptr0++,outgradient[a]);
      fputc(*ptr0,outgradient[a]);
    }
    }
  fclose(outgradient[a]);
}

/*---------------------------------------------------------------------------*/
/*  Smoothing and export of smoothed pictures onto files                     */
/*---------------------------------------------------------------------------*/

void smoothedpic(int a){                          // write smooth picture into inital input bin
char *inptr,*outptr;
int i,j,ii,jj,u,v,k;
long r2,g2,b2;
int r3,g3,b3;
  for(j = 0; j > -y_size; j--){
  for(i = 0; i <  x_size; i++){
      for (u=-2; u<=2; u++) {
      for (v=-2; v<=2; v++) {
        ii=xwrap(i+u);jj=-ywrap(-j+v);
        inptr=outputarray[a]+3*((-jj)*x_size+ii);
        rpatch[u+2][v+2]=chartoint(*inptr++);gpatch[u+2][v+2]=chartoint(*inptr++);bpatch[u+2][v+2]=chartoint(*inptr);
      }
      }
      r2=0; g2=0; b2=0;
      for (u=0; u<=4; u++) {
      for (v=0; v<=4; v++) {
        r2 += gaussian[u][v]*rpatch[u][v]; g2+=gaussian[u][v]*gpatch[u][v]; b2+=gaussian[u][v]*bpatch[u][v];
      }
      }
      r3 = floor(r2/gaussiansum); g3 = floor(g2/gaussiansum); b3 = floor(b2/gaussiansum);
      outptr    = inputarray[a] + 3*((-j)*x_size+i );
      *outptr++ = r3; *outptr++ = g3; *outptr++ = b3;
  }
  }
}

/*---------------------------------------------------------------------------*/
/*  Curvature computation and writing out to files                           */
/*---------------------------------------------------------------------------*/
```

```
void compute_curvature(int a, int b){         // write curvature picture into allocated curvature memory
char *inptr,*curvatureptr,*gradientptr;
int i,j,ii,jj,u,v,k;
int  r2,g2,b2;
int  r3,g3,b3;
int  newrmaxthresh,newgmaxthresh,newbmaxthresh;     // adapt the thresholds
int  newrmaxedge,newgmaxedge,newbmaxedge;

fprintf(stderr,"Compute curvatures and gradients:");
for (k=a; k<=b; k++) {
  fprintf(stderr," %i ",k);
  newrmaxthresh=1; newgmaxthresh=1; newbmaxthresh=1;
  newrmaxedge=1;   newgmaxedge=1;   newbmaxedge=1;
  for(j = 0; j > -y_size; j--){
  for(i = 0; i <  x_size; i++){
        curvatureptr    = outcurvaturearray[k] + 3*((-j)*x_size+i );
        gradientptr     = outgradientarray[k]  + 3*((-j)*x_size+i );
        if ( (j > (-y_size+bb)) && (j<-bb) && (i < (x_size-bb)) && (i>bb) ){
          curvature1(k,i,j);
          r2 = floor((255.0*rthresh)/maxrthresh[k]); g2 = floor((255.0*gthresh)/maxgthresh[k]); b2 = floor((255.0*bthresh)/maxbthresh[k]);
          r3 = floor((255.0*redge)/maxredge[k]);     g3 = floor((255.0*gedge)/maxgedge[k]);     b3 = floor((255.0*bedge)/maxbedge[k]);
          *gradientptr++  = r2;  *gradientptr++  = g2; *gradientptr++ = b2;
          *curvatureptr++ = r3;  *curvatureptr++ = g3; *curvatureptr++ = b3;
        } else  {
          *curvatureptr++ = 0; *curvatureptr++ = 0; *curvatureptr++ = 0;
          *gradientptr++  = 0; *gradientptr++  = 0; *gradientptr++  = 0;
        }
       if (rthresh>newrmaxthresh) { newrmaxthresh=rthresh; }
       if (gthresh>newgmaxthresh) { newgmaxthresh=gthresh; }
       if (bthresh>newbmaxthresh) { newbmaxthresh=bthresh; }
       if (redge>newrmaxedge) { newrmaxedge=redge; }
       if (gedge>newgmaxedge) { newgmaxedge=gedge; }
       if (bedge>newbmaxedge) { newbmaxedge=bedge; }
  }
  }
  if (k<b) {
    maxrthresh[k+1]=newrmaxthresh; maxgthresh[k+1]=newgmaxthresh; maxbthresh[k+1]=newbmaxthresh;
    maxredge[k+1]=newrmaxedge; maxgedge[k+1]=newgmaxedge; maxbedge[k+1]=newbmaxedge;
  }

  write_curvature(k);
  write_gradient(k);
}
fprintf(stderr,"\n");
}

/*---------------------------------------------------------------------------*/
/*  Shuffling around some memory  for example to iterate smoothing           */
/*---------------------------------------------------------------------------*/

void copypic(int a){                  // copy the picture from input to output
char *inptr,*outptr;
int i,j,k;
  for(j = 0; j > -y_size; j--){
  for(i = 0; i <  x_size; i++){
      inptr     = inputarray[a]  + 3*((-j)*x_size+i );
      outptr    = outputarray[a] + 3*((-j)*x_size+i );
      *outptr++ = *inptr++; *outptr++ = *inptr++; *outptr++ = *inptr++;
  }
  }
}

void copyallpic(int a, int b){
int k;
for (k=a; k<=b; k++) { copypic(k); }
}

void iterated_smoothedpic(int a, int b){
int k,l;
fprintf(stderr,"Produce smoothed pictures:  ");
for (l=a; l<=b; l++) {
   fprintf(stderr," %i ",l);
   for (k=0; k<S; k++) {                       // smooth S times, switching forth and back from input to output
        smoothedpic(l);
        copypic(l);
    }
   write_smoothed(l);
}
fprintf(stderr,"\n");
}


/*-------------------------------------------------------------------------------*/
/*  End in various files                                                         */
/*-------------------------------------------------------------------------------*/


void readpics(int a, int b){
char buffer[1025];
char *ptr;
int i,k;
  fprintf(stderr,"Read in original pictures:");
  for (k=a; k<=b; k++) {
    fprintf(stderr,".");
    fgets(buffer,1025,in[k]);
    if(strncmp(buffer,"P6",2)){ fprintf(stderr,"Error (need PPM raw)\n"); exit(1); }
    do fgets(buffer,1025,in[k]); while(*buffer == '#');
    x_size = atoi(strtok(buffer," ")); y_size = atoi(strtok(NULL," "));
    fgets(buffer,1025,in[k]); c_size = atoi(buffer);
    if((inputarray[k]= (char *) malloc(3*x_size*y_size*sizeof(char)))==NULL){ fprintf(stderr,"Err\n"); exit(1);}
    i = 0; ptr = inputarray[k];
    while(!feof(in[k]) && i<3*x_size*y_size){ *ptr++ = fgetc(in[k]); i++;} fclose(in[k]);
    if(i<x_size*y_size){ fprintf(stderr,"Premature end of image\n"); exit(1); }
  }
  fprintf(stderr,"\n");
}

void readsmoothedpics(int a, int b){
char buffer[1025];
char *ptr;
int i,k;
  if (preprocessed==1) {                       // only if preprocessing has happened
  fprintf(stderr,"Read in smoothed pictures:");
  for (k=a; k<=b; k++) {
```

```
    fprintf(stderr,".");
    fgets(buffer,1025,insmoothed[k]);
    if(strncmp(buffer,"P6",2)){ fprintf(stderr,"Error (need PPM raw)\n"); exit(1); }
    do fgets(buffer,1025,insmoothed[k]); while(*buffer == '#');
    x_size = atoi(strtok(buffer," ")); y_size = atoi(strtok(NULL," "));
    fgets(buffer,1025,insmoothed[k]); c_size = atoi(buffer);
    if((insmoothedarray[k]= (char *) malloc(3*x_size*y_size*sizeof(char)))==NULL){ fprintf(stderr,"Err\n"); exit(1);}
    i = 0; ptr = insmoothedarray[k];
    while(!feof(insmoothed[k]) && i<3*x_size*y_size){ *ptr++ = fgetc(insmoothed[k]); i++;} fclose(insmoothed[k]);
    if(i<x_size*y_size){ fprintf(stderr,"Premature end of image\n"); exit(1); }
  }
  }
  fprintf(stderr,"\n");
}

void readcurvaturepics(int a, int b){
char buffer[1025];
char *ptr;
int i,k;
  if (preprocessed==1) {                       // only if preprocessing has happened
  fprintf(stderr,"Read in curvature pictures:");
  for (k=a; k<=b; k++) {
    fprintf(stderr,".");
    fgets(buffer,1025,incurvature[k]);
    if(strncmp(buffer,"P6",2)){ fprintf(stderr,"Error (need PPM raw)\n"); exit(1); }
    do fgets(buffer,1025,incurvature[k]); while(*buffer == '#');
    x_size = atoi(strtok(buffer," ")); y_size = atoi(strtok(NULL," "));
    fgets(buffer,1025,incurvature[k]); c_size = atoi(buffer);
    if((incurvaturearray[k]= (char *) malloc(3*x_size*y_size*sizeof(char)))==NULL){ fprintf(stderr,"Err\n"); exit(1);}
    i = 0; ptr = incurvaturearray[k];
    while(!feof(incurvature[k]) && i<3*x_size*y_size){ *ptr++ = fgetc(incurvature[k]); i++;} fclose(incurvature[k]);
    if(i<x_size*y_size){ fprintf(stderr,"Premature end of image\n"); exit(1); }
  }
  }
  fprintf(stderr,"\n");
}

void readgradientpics(int a, int b){
char buffer[1025];
char *ptr;
int i,k;
  if (preprocessed==1) {                       // only if preprocessing has happened
  fprintf(stderr,"Read in gradient pictures:");
  for (k=a; k<=b; k++) {
    fprintf(stderr,".");
    fgets(buffer,1025,ingradient[k]);
    if(strncmp(buffer,"P6",2)){ fprintf(stderr,"Error (need PPM raw)\n"); exit(1); }
    do fgets(buffer,1025,ingradient[k]); while(*buffer == '#');
    x_size = atoi(strtok(buffer," ")); y_size = atoi(strtok(NULL," "));
    fgets(buffer,1025,ingradient[k]); c_size = atoi(buffer);
    if((ingradientarray[k]= (char *) malloc(3*x_size*y_size*sizeof(char)))==NULL){ fprintf(stderr,"Err\n"); exit(1);}
    i = 0; ptr = ingradientarray[k];
    while(!feof(ingradient[k]) && i<3*x_size*y_size){ *ptr++ = fgetc(ingradient[k]); i++;} fclose(ingradient[k]);
    if(i<x_size*y_size){ fprintf(stderr,"Premature end of image\n"); exit(1); }
  }
  }
  fprintf(stderr,"\n");
```

```
}

/*--------------------------------------------------------------------------*/
/*  Memory allocation for various picture arrays (movies)                   */
/*--------------------------------------------------------------------------*/


void memoryalloc(int a, int b){
int k;
fprintf(stderr,"Allocate memory:\n");
  for (k=a; k<=b; k++) {
    if((outputarray[k]       = (char *) malloc(3*x_size*y_size*sizeof(char)))==NULL){fprintf(stderr,"Err\n");exit(1);}
    if((outcurvaturearray[k] = (char *) malloc(3*x_size*y_size*sizeof(char)))==NULL){fprintf(stderr,"Err\n");exit(1);}
    if((outgradientarray[k]  = (char *) malloc(3*x_size*y_size*sizeof(char)))==NULL){fprintf(stderr,"Err\n");exit(1);}
  }
}

/*--------------------------------------------------------------------------*/
/*  Merging points is nifty but makes export of paths a bit tricky          */
/*--------------------------------------------------------------------------*/


void mergepoints(int a){                    // merge points with similar position and velocity
                                            // and where curvature values have not deteriorated a lot
int cp, newcp, cpl;                         // countpoint index
int take;                                   // take the point (1) or not (0)
int  x1_match[MP], y1_match[MP];            // point position
int  v1_match[MP], w1_match[MP];            // point velocity
double  thresh,edge;                        // average over colors


for (cp=0; cp<countpoints[a]; cp++)  {      // copy points over and compute velocities
   x1_match[cp]=x_match[cp][a];
   y1_match[cp]=y_match[cp][a];
   v1_match[cp]=x_match[cp][a-1];
   w1_match[cp]=y_match[cp][a-1];
}

newcp=0;
for (cp=0; cp<countpoints[a]; cp++) {
    take=1; cpl=0;
    curvature(a,x1_match[cp],y1_match[cp]);
    thresh=(rthresh+gthresh+bthresh)/3; edge=(redge+gedge+bedge)/3;
    if  ( thresh<threshold/3  || edge < ethreshold/3 )  { take =0; }
    while (cpl<newcp && take==1) {
       if (
           ( abs(x1_match[cpl]-x_match[cp][a])<merge) &&
           ( abs(y1_match[cpl]-y_match[cp][a])<merge) &&
           ( abs(v1_match[cpl]-x_match[cp][a-1])<merge) &&
           ( abs(w1_match[cpl]-y_match[cp][a-1])<merge)
         ) { take=0; } cpl++;
    }
    if (take==1) {
      x_match[newcp][a]=x1_match[cp]; y_match[newcp][a]=y1_match[cp];   newcp++;
    }
}
countpoints[a]=newcp;
```

```
fprintf(stderr,"New number of points for %i frame: %i\n", a,countpoints[a]);
}

/*--------------------------------------------------------------------------*/
/*  Write the tracked points onto a file                                    */
/*--------------------------------------------------------------------------*/

void write_points(int aa1, int bb1){
FILE *res;                                  // outputfile
int  k,cp;                                  // index for counting points
char *match_file="points.dat";              // name of output file
  res=fopen(match_file,"w");
  fprintf(res,"0 0 0 0 %i %i \n",x_size,y_size);    // print first the size of the picture
  for (k=aa1; k<=bb1; k++) {
    for (cp=1; cp<countpoints[k]; cp++) { // camera number k, first frame aa1, last frame bb1, point number k, xcoord, ycoord
      fprintf(res,"%i %i %i %i %i %i \n",k,aa1,bb1,cp,x_match[cp][k],-y_match[cp][k]);
    }
  }
  fclose(res);
}

/*--------------------------------------------------------------------------*/
/*  Various initializations like file names or checks that pictures exist    */
/*--------------------------------------------------------------------------*/

void initializations(int a, int b) {
int k;
if (b>NF) {fprintf(stderr,"Use less pictures\n\n"); }
if (fopen("tmp/c1000.pnm","rb")==NULL) { preprocessed=0; } else { preprocessed=1; }
if (fopen("tmp/s1000.pnm","rb")==NULL) { smootheddone=0; } else { smootheddone=1; }

for (k=a; k<=b; k++) {
    outfilename[k]      =make_filename_a(1000+k,"pnm");
    smoothfilename[k]   =make_filename_s(1000+k,"pnm");
    curvaturefilename[k]=make_filename_c(1000+k,"pnm");
    gradientfilename[k] =make_filename_g(1000+k,"pnm");

    if((in[k] = fopen(infilename[k],"rb")) == NULL){              // check  for input files
        fprintf(stderr,"Error %s\n", infilename[k]); exit(1);
    }

    if (preprocessed==1) {                                        // check for curvature files
      if((incurvature[k] = fopen(curvaturefilename[k],"rb")) == NULL){
         fprintf(stderr,"Error %s\n", curvaturefilename[k]); exit(1);
      }

                                                                 //  smoothed files
      if((insmoothed[k] = fopen(smoothfilename[k],"rb")) == NULL){
         fprintf(stderr,"Error %s\n", smoothfilename[k]); exit(1);
      }

                                                                  //  ... and gradient files
      if((ingradient[k] = fopen(gradientfilename[k],"rb")) == NULL){
         fprintf(stderr,"Error %s\n", gradientfilename[k]); exit(1);
      }

    }
}
}

/*-----------------------------------------------------------------------------*/
/*  Do preprosessing                                                           */
/*-----------------------------------------------------------------------------*/

void smoothpictures(int a, int b) {
int k;

if (smootheddone==0) {                          // preprocessing is necessary
    iterated_smoothedpic(a,b);                  // computed smooth pictures
} else { fprintf(stderr,"No smoothing necessary\n");  }
}


void preprocesscurvature(int a, int b) {
int k;
if (preprocessed==0) {                          // preprocessing is necessary
    find_parameters(a);                         // determine parameters from first picture
    for (k=a+1; k<=b; k++) {
         maxrthresh[k]= maxrthresh[a]; maxgthresh[k]= maxgthresh[a]; maxbthresh[k]= maxbthresh[a];
         maxredge[k]  = maxredge[a];   maxgedge[k]  = maxgedge[a];   maxbedge[k]  = maxbedge[a];
    }
    compute_curvature(a,b);                     // compute the curvatures
  } else { fprintf(stderr,"No curvature computations necessary\n"); }
}

void processvectorfields(int a, int b) {        // the main tracking procedure
int k;
if (preprocessed==1) {
  initial_vectorfield(a+1);                     // start vector field and point choice
  for (k=a+2; k<b; k++) {
     vectorfield(k);
     countpoints[k]  =countpoints[k-1];
     countpoints[k+1]=countpoints[k-1];
     // mergepoints(k);
     write_output(k);
  }
  fprintf(stderr,"\n");
}
}

/*-----------------------------------------------------------------------------*/
/*  The main file                                                              */
/*-----------------------------------------------------------------------------*/

int main(int argc, char *argv[]){
int a,b,k;
  a=0; b=argc-2;
  if(b<=NF) { for (k=a; k<=b; k++) { infilename[k]=argv[k+1]; } }
  initializations(a,b);                       // make filenames
  readpics(a,b);                              // read in the pictures
  readsmoothedpics(a,b);                      // read in the smoothed pictures if exist
  readcurvaturepics(a,b);                     // read in the curvature pictures if exist
  readgradientpics(a,b);                      // read in the gradient pictures if existent
  memoryalloc(a,b);                           // memory allocation
```

```
  definematrices();                           // define various matrices
  copyallpic(a,b);                            // copy pictures over
  smoothpictures(a,b);                        // smooth pictures if necessary
  preprocesscurvature(a,b);                   // compute curvature values if necessary
  processvectorfields(a,b);                   // track the points along the vector fields
  write_points(a+3,b-1);                      // write points to text file
  write_output(1);                            // write output file with trajectories
  return 0;
}

/*-----------------------------------------------------------------------------*/
/*  The end                                                                    */
/*-----------------------------------------------------------------------------*/




/*-----------------------------------------------------------------------------*/
/*  statistics.m                                                              */
/*-----------------------------------------------------------------------------*/

/*\
 *  This file is part of the Omnivis project, Harvard University
 *  Copyright (C) December 2009  Oliver Knill and Jose Ramirez Herran
 *  http://www.math.harvard.edu/~knill/3dscan2           (project page)
 *
 *  Usage:  math<statistics.m>out
 *  The program needs to read an imput file called out, which contains the angles.
 *  This is the mathematica implementation of Structure from Motion
 *  for omnidirectional vision. The output is a rendered scene of the points
 *  reconstructed.
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
\*/
MMM=3;    (* number of world points  add two more points  *)
NNN=3;    (* number of cameras      *)
P={{0,0,0},{1,0,0},{3,3/2,0}}                                                          (* world points              *)
Q={{-1,2,0},{1/2,3,0},{2,5/2,0}};                                                      (* camera points             *)

m=Length[P];  n=Length[Q];                                                             (* number of points and obser*)
ActualPlanarData=Flatten[{Table[Q[[j,1]],{j,n}],Table[Q[[j,2]],{j,n}],
                          Table[P[[i,1]],{i,3,m}],Table[P[[i,2]],{i,2,m}]}];           (* write planar data as vect *)
ActualSpaceData=Flatten[{Table[Q[[j,3]],{j,n}],Table[P[[i,3]],{i,2,m}]}];              (* write space  data as vect *)
L[Aa_,Bb_]:= Arg[(Aa[[1]]-Bb[[1]]) + I (Aa[[2]]-Bb[[2]])];                             (* theta angle               *)
Obs= Table[L[P[[i]],Q[[j]]],{i,m},{j,n}];                                              (* list of angles            *)
```

```
RR = Table[N[Sqrt[(P[[i,1]]-Q[[j,1]])^2 + (P[[i,2]]-Q[[j,2]])^2]],{i,m},{j,n}];        (* distances                 *)
ZZ = Table[P[[i,3]]-Q[[j,3]],{i,m},{j,n}];                                              (* z coordinates of world    *)
Obs1=Table[Arg[I ZZ[[i,j]] + RR[[i,j]]],{i,m},{j,n}];                                   (* list of vertical slopes   *)



m = Length[Obs];                                                                        (* number of world points    *)
n = Length[Obs[[1]]];                                                                   (* number of camera points   *)

O1C =Table[Cos[Obs1[[i,j]]],{i,m},{j,n}];                                               (* cos(phi)                  *)
O1S =Table[Sin[Obs1[[i,j]]],{i,m},{j,n}];                                               (* sin(phi)                  *)
OC=  Table[Cos[Obs[[i,j]]],{i,m},{j,n}];                                                (* cos(theta)                *)
OS=  Table[Sin[Obs[[i,j]]],{i,m},{j,n}];                                                (* sin(theta)                *)
PP=Table[{x[i],y[i],z[i]},{i,m}];                                                       (* variables for points      *)
QQ=Table[{a[j],b[j],c[j]},{j,n}];                                                       (* variables for camera      *)



Eq=Flatten[Table[OC[[i,j]](PP[[i,2]]-QQ[[j,2]])-OS[[i,j]](PP[[i,1]]-QQ[[j,1]]),{i,m},{j,n}]];  (* the equations     *)
Va=Union[Flatten[Table[{x[i],y[i],a[j],b[j]},{i,m},{j,n}]]];                            (* all variables as 1 vector *)
mm=Length[Eq];  nn=Length[Va];                                                          (* number of variables       *)
A=Table[D[Eq[[i]],Va[[j]]],{i,mm},{j,nn}];                                              (* matrix of the system      *)
AT=Transpose[A];                                                                        (* its transpose for columns *)
x1pos=Position[Va,x[1]][[1,1]];                                                         (* row containing x[1]       *)
x2pos=Position[Va,x[2]][[1,1]];                                                         (* row containing x[2]       *)
y1pos=Position[Va,y[1]][[1,1]];                                                         (* row containing y[1]       *)
bbb=-AT[[x2pos]];                                                                       (* x2=1 gives inhomog. part  *)
ss=Table[k,{k,nn}];                                                                     (* list of rows              *)
ss1=Complement[ss,{x1pos,x2pos,y1pos}];                                                 (* normalize x1=y1=z1=0,x2=1 *)
AT1=Table[AT[[ss1[[k]]]],{k,Length[ss1]}];                                              (* Without x1,x2,y1 row      *)
A1=Transpose[AT1];                                                                      (* Without x1,x2,y1 column   *)
ComputedPlanarData=Inverse[AT1.A1].AT1.bbb;                                             (* A x = b by (A^TA)^-1 A^Tb *)
Max[Abs[ActualPlanarData-ComputedPlanarData]];                                          (* maximal error             *)
(* Now we have the (x_i,y_i,a_i,b_i) coordinates. Now we start the 3d reconstruction                                 *)
aa=Table[ComputedPlanarData[[j]],{j,n}];                                                (* x-positions camera points *)
bb=Table[ComputedPlanarData[[j]],{j,n+1,2n}];                                           (* y-positions camera points *)
xx=Flatten[{{0,1},Table[ComputedPlanarData[[j]],{j,2n+1,2n+1+m-3}]}];                   (* x-position of world       *)
yy=Flatten[{{0},Table[ComputedPlanarData[[j]],{j,2n+m-1,2n+2m-3}]}];                    (* y-position of world       *)
rr=Table[N[Sqrt[(xx[[i]]-aa[[j]])^2 + (yy[[i]]-bb[[j]])^2]],{i,m},{j,n}];               (* distances                 *)
Va1=Union[Flatten[Table[{z[i],c[j]},{i,m},{j,n}]]];                                     (* all variables as 1 vector *)
Eq1=Flatten[Table[O1C[[i,j]](PP[[i,3]]-QQ[[j,3]])-O1S[[i,j]] rr[[i,j]],{i,m},{j,n}]];   (* the equations             *)
mm1=Length[Eq1];  nn1=Length[Va1];                                                      (* number of eq and variab   *)
A =Table[D[Eq1[[i]],Va1[[j]]],{i,mm1},{j,nn1}];                                         (* matrix of the system      *)
AT=Transpose[A];                                                                        (* its transpose for columns *)
z1pos=Position[Va1,z[1]][[1,1]];                                                        (* row containing z[1]       *)
ss=Table[k,{k,nn1}];                                                                    (* list of rows              *)
ss1=Complement[ss,{z1pos}];                                                             (* normalize z1=0            *)
AT1=Table[AT[[ss1[[k]]]],{k,Length[ss1]}];                                              (* Without z1 row            *)
A1=Transpose[AT1];                                                                      (* Without z1 column         *)
b1 = -Table[Eq1[[i]]-Sum[A[[i,j]] Va1[[j]],{j,Length[Va1]}],{i,Length[Eq1]}];           (* the inhomogenous part     *)
b1 = Chop[Simplify[b1]];                                                                (* remove                    *)
ComputedSpaceData=Inverse[AT1.A1].AT1.b1;                                               (* A x = b by (A^TA)^-1 A^Tb *)



/*---------------------------------------------------------------------------*/
/*  translate.m                                                              */
/*---------------------------------------------------------------------------*/
```

```
/*\
 *  This file is part of the Omnivis project, Harvard University
 *  Copyright (C) December 2009  Oliver Knill and Jose Ramirez Herran
 *  http://www.math.harvard.edu/~knill/3dscan2           (project page)
 *
 *  Usage:  math<translate.m>out
 *  The program needs to read an imput file called points.dat, which contains
 *  the pixel coordinates of the corners tracked.
 *  This is the mathematica implementation of the coordinate transformation
 *  for omnidirectional vision. The output is the angle information neededed
 *  for the statistics.m program
 *
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
\*/

(* this is an interface to process data which had been computed by the C program from the movie                *)
(* What is done here should better be done in C in future. Mathematica is too slow for such things             *)
(* This program selects interesting points and writes them onto a file out.m, which will be consequently       *)
(* be read by the reconstruction program                   O. Knill, June-July 2009                            *)
(* points.dot contains data     Point#   FirstFrame  LastFrame   Time    Xcoord   Ycoord                       *)

(* we start with two numbers which are crucial. These numbers are used to thin out the data points             *)

fr=0.7;                                                   (* if fr*maxfluctuations, we take a point         *)
fm=0.00001;                                               (* if fluctuations < fm*maxfluctuations, discard  *)

(* read in the data file written by the C program                                                              *)
A = ReadList["points.dat", "Number"];
(* A = ReadList["/Users/knill/art-3dscan2/c/correspondence15/points.dat","Number"];     (* For OS X Laptop *)  *)

(* organize the raw data                                                                                       *)

B=Partition[A,6]; width=B[[1,5]]; height=B[[1,6]];
B1=Table[B[[k]],{k,2,Length[B]}]; B=B1; bb=Length[B];   (* cut away the first line, which contains geometry    *)

(* Renumber due to start with frame 3  *)

B2 = Table[ {B[[k,1]]-2, B[[k,2]]-2,B[[k,3]]-2,B[[k,4]],B[[k,5]],B[[k,6]]},{k,Length[B]}]; B=B2;

xmax=Max[Table[B[[i,5]],{i,bb}]];                       (* maximal x and y coordinates                         *)
ymax=Max[Table[B[[i,6]],{i,bb}]];
fmax=Last[B][[3]];                                      (* number of time steps in total                       *)
BNr=Table[B[[i,2]]*1000^2 + B[[i,3]]*1000 + B[[i,4]],{i,bb}];  PointLabels=Union[BNr];
```

```
pnumber=Length[PointLabels];   (* number of different points tracked *)
orbitlengths=Table[Length[Position[BNr, PointLabels[[k]]]], {k, pnumber}];  (* list of orbit lengths for pnt k *)
tmax = Max[orbitlengths];                                (* this is the maximal orbit length                    *)

(* Procedures to access individual points                                                                       *)

OrbitLength[k_]:=orbitlengths[[k]];                              (* orbit length of point k                    *)
PointInterval[k_]:=Flatten[Position[BNr,PointLabels[[k]]]];      (* list of array indices dealing with point k *)
Info[k_]:=Module[{}, a=PointInterval[k]; Table[B[[a[[l]]]],{l,Length[a]}]];
Positions[k_]:=Module[{},b=Info[k]; Table[{b[[l,5]],b[[l,6]]},{l,Length[b]}]];
StartTime[k_]:=Module[{},b=First[Info[k]]; b[[1]]];              (* first time the point is tracked            *)
EndTime[k_]:=Module[{},b=Last[Info[k]]; b[[1]]];                 (* last  time the point is tracked            *)
PlotPointRaw[k_]:=Graphics[Line[Positions[k]]];
XpixelToAngle[xp_]:= N[2Pi xp/width];                            (* theta angle                                *)
YpixelToAngle[yp_]:= N[Pi (yp-height/2)/height];                 (* phi angle                                  *)
AnglePositions[k_]:=Module[{},b=Info[k]; Table[{XpixelToAngle[b[[l,5]]],YpixelToAngle[b[[l,6]]]},{l,Length[b]}]];
PlotPoint[k_]:=Show[Graphics[Line[AnglePositions[k]],PlotRange->{{0,2Pi},{-Pi/2,Pi/2}}]];
AllOrbitlengths = Table[OrbitLength[kk],  {kk, pnumber}];
Fluctuations[k_]:=Module[{},pos=AnglePositions[k];               (* discrete second derivatives (accelaration) *)
   Sum[Norm[pos[[j+2]]-2 pos[[j+1]]+pos[[j]]],{j,Length[pos]-2}]];

(* collect statistics  *)

AllFluctuations = Table[Fluctuations[kk], {kk, pnumber}];               (* this takes time to compute   *)
MaxFluctuations = Max[AllFluctuations];
IllustrateFluctuations := Show[Graphics[Table[Point[{k, AllFluctuations[[k]]}],{k, pnumber}]],AspectRatio -> 1];

(* Selecting interesting points. Want fluctuations small, possibly from all parts of picture            *)
(* the later requirement is not yet implemented                                                         *)

Obs = Table[ 0,{i,1,pnumber},{j,1,fmax}];  Obs1=Obs;  Mask=Obs;    (* empty to be filled    *)
numberofpoints=0; numberofcameras=0;                               (* we need to count how many points  *)
minfluctuations=fm*MaxFluctuations;                                (* lower bound for fluctuations      *)
maxfluctuations=fr*MaxFluctuations;                                (* where to take the cut for flucts  *)

Do[
   orbitinfo   = AnglePositions[k];             (* angle information about the actual point k           *)
   fluctuations= AllFluctuations[[k]];          (* the acceleration fluctuation of the point k          *)
   starttime  = StartTime[k];                   (* first time the orbit is tracked                      *)
   numberofpoints++;                            (* we have found a good point and count up              *)
   Do[
      time = starttime+l-1;  numberofcameras=Max[numberofcameras,time];
      Obs[[ numberofpoints, time]]=orbitinfo[[l,1]];        (* first angle                              *)
      Obs1[[numberofpoints, time]]=orbitinfo[[l,2]];        (* second angle                             *)
      Mask[[numberofpoints, time]] = 1, {l,1,Length[orbitinfo]}],
   {k,1,pnumber}]; Print[numberofpoints];

(* write out the results to a file "out.m"                                                              *)

Run["touch out.m; rm out.m"];                  (* in order not to add just to a previously written file *)
WriteString["out.m"," "];
WriteString["out.m","Obs={"];
Do[ WriteString["out.m","{"];
    Do[ WriteString["out.m",Obs[[k,l]]]; If[l<numberofcameras,WriteString["out.m",","]],{l,numberofcameras}];
    WriteString["out.m","}"];            If[k<numberofpoints,WriteString["out.m",",\n"]],{k,numberofpoints}];
WriteString["out.m","};\n"];
```

```
WriteString["out.m","Mask={"];
Do[ WriteString["out.m","{"];
    Do[ WriteString["out.m",Mask[[k,l]]]; If[l<numberofcameras,WriteString["out.m",","]],{l,numberofcameras}];
    WriteString["out.m","}"];             If[k<numberofpoints,WriteString["out.m",",\n"]],{k,numberofpoints}];
WriteString["out.m","};\n"];

WriteString["out.m","Obs1={"];
Do[ WriteString["out.m","{"];
    Do[ WriteString["out.m",Obs1[[k,l]]]; If[l<numberofcameras,WriteString["out.m",","]],{l,numberofcameras}];
    WriteString["out.m","}"];             If[k<numberofpoints,WriteString["out.m",",\n"]],{k,numberofpoints}];
WriteString["out.m","};"];




/*----------------------------------------------------------------------------*/
/*  plotpoints.m                                                              */
/*----------------------------------------------------------------------------*/

/*\
 *  This file is part of the Omnivis project, Harvard University
 *  Copyright (C) December 2009  Oliver Knill and Jose Ramirez Herran
 *  http://www.math.harvard.edu/~knill/3dscan2           (project page)
 *
 *  Usage:  math<plotpoints.m
 *  The program needs to read an imput file called points.dat, which contains
 *  the pixel coordinates of the corners tracked.
 *  This is the mathematica implementation for visualizing the points for
 *  debugging purposes and visualization of the input data. The output
 *  is an image of the path of each point tracked to be used in the
 *  reconstruction.
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
\*/
(* ::Package:: *)

(* just print the points  *)

A = ReadList["points.dat", "Number"];
(* A = ReadList["/Users/knill/art-3dscan2/c/correspondence13/points.dat","Number"]; *)

B = Partition[A, 6]; width = B[[1, 5]]; height = B[[1, 6]];
B1 = Table[B[[k]], {k, 2, Length[B]}]; B = B1; bb = Length[B];
xmax = Max[Table[B[[i, 5]], {i, bb}]]; ymax = Max[Table[B[[i, 6]], {i, bb}]];
tmax = Max[Table[B[[i, 1]], {i, bb}]] + 1; pmax = Max[Table[B[[i, 4]], {i, bb}]] + 1;
```

```
FindPoints[startframe_,pointnr_]:=Module[{},s={};
     Do[If[B[[i,2]]==startframe && B[[i,4]]==pointnr && B[[i,6]]>0 && B[[i,6]]<ymax,
     s=Append[s,{B[[i,5]],height-B[[i,6]]}]],{i,3,Length[B]-2}];s];
PlotOrbit[startframe_,pointnr_]:=Module[{},
     u=FindPoints[startframe,pointnr]; {Hue[pointnr/pmax],Line[u]}];
S=Show[Graphics[Table[PlotOrbit[3,k],{k,1,pmax-1}]]]

Export["3d.png",S,"PNG",ImageSize->1200]


/*--------------------------------------------------------------------------*/
/*  Makefile.m                                                              */
/*--------------------------------------------------------------------------*/
*\
 *  This file is part of the Omnivis project, Harvard University
 *  Copyright (C) December 2009  Oliver Knill and Jose Ramirez Herran
 *  http://www.math.harvard.edu/~knill/3dscan2          (project page)
 *
 *  Usage:  make math
 *  This program runs our mathematica implementation of Structure from Motion
 *  for omnidirectional vision. The output is a rendered scene of the points
 *  reconstructed. Is a script file to use in linux/unix systems.
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
\*/

math:
touch newpoints.m; rm newpoints.m
math<translate.m>out
math<statistics.m>out
xv firstrun.png
```

Index

3D reconstruction, i

Algebraic model for fast corner detection, 32

brightness, 19

camera calibration, 7
catadioptric, 40
catadioptric
    Omnidirectional vision sensor, 3
catadioptric sensor, i
color image, 103
computations
    low-level vision, 17
Computer vision, i, 2, 15
Conical mirror, 37
Corner detection, i
corner detection, 21
Corner detection
    algebraic , 32
corner detection
    Harris, 28
    Kitchen-Rosenfeld, 24
    SUSAN, 32
corner detector
    cross , 24
corner tracking
    KLT, 24
cornerness, 29
curvature, 27

derivative
    Sobel, 23

edge detection, 22

Feature
    detector, i
    extraction, i
    tracking, i
feature extraction, 21
feature tracking, 68
Features
    FAST, 9
    Generalized Voronoi Diagram, 9
    SIFT, 9
    SURF, 9
features
    low level, 17

frame, 16
frame rate, 16

Gaussian, 21
Gigantocypris, 40
Google
  Earth, 2
  street view, 2
gray scale image, 103

Harris corner detection, 28
Harris detector, 29
histogram, 18
Hyperbolic mirror, 38

image
  black and white, 103
  color, 103
  gray, 103
Image format , JPG 16, PNG 16, PNM 16, PS 16
image processing, 15, 16
inter-reflections, 22

Kitchen-Rosenfeld corner detection, 24

Laplacian, 22
least squares, 5
Linear least squares, i
Low-level vision computations, 17

Mirror
  hyperboloidal, 9
  parabolic, i, 9
  spherical, 9
Moravec detector, 29
motion tracking, 7
movie, 16
Multiple camera array
  planar mirrors, 9
  spherical, 9

off-line processing, i
Omnidirectional
  camera, i, 3, 5
  images, 4
  video, i
  vision, i

Panorama, 4
panoramic camera, 10
panoramic images, 5
perceptual computer interface, 7
Persistence Of Vision ray tracer, 5
Perspective
  images, 4
pixel, 17
POV-Ray
  camera, 5
  movie, 5
Process
  Off-line, 2
  Offline, 9
  On-line, 2
Proposed pipeline, 91

robot navigation, 7, 48

semi synthetic
  environment, 7
  reconstruction, 5
shadows, 22
Sobel derivative, 23
Sobel partial derivative, 23
specular reflections, 22
Spherical Omnidirectional mirror, 37
Structure from motion, i, 1, 2, 5, 7–9

thresholding, 18, 20
Tracking
  PROSAC, 9
  system, i
tracking
  feature, 68

Unwrapping, 3, 41

Zero crossings, 23