Final Year Project Report

Object Detection and Tracking in Images and Point Clouds

Daniel J. Finnegan

A thesis submitted in part fulfilment of the degree of

BA/BSc (hons) in Computer Science

Supervisor: Dr. Eleni Mangina

Moderator: Dr. Mauro Dragone



UCD School of Computer Science and Informatics College of Engineering Mathematical and Physical Sciences University College Dublin

March 24, 2012

Table of Contents

_

Ab	strac	t	3							
1	In	$\operatorname{troduction}$	5							
	1.1	Preamble	5							
	1.2	Initial Specification & Overview	5							
	1.3	Tracking	6							
	1.4	Justification	6							
	1.5	Roadmap	7							
2	Ba	ckground Research	8							
	2.1	Preamble	8							
	2.2	Technology	8							
	2.3	Algorithms and Research	9							
3	De	Design								
	3.1	Preamble	12							
	3.2	Design Specification	12							
	3.3	Software Practices	13							
	3.4	Work Breakdown Structure	14							
4	\mathbf{P}	roject Implementation	15							
	4.1	Preamble	15							
	4.2	System Overview	15							
	4.3	2D Tracking Using OpenCV	16							
	4.4	3D Visualization and Augmented Tracking With PCL	18							
	4.5	Message Passing in ROS	19							
	4.6	Summary	19							
5	R	esults & Conclusions	21							
	5.1	Preamble	21							
	5.2	Performance Benchmarking	21							

	5.3	Critical Analysis	23
	5.4	Conclusions	23
6	Fι	uture Work	25
	6.1	Autonomous Control of a Robot	25
	6.2	Optimisations and Distribution	25
	6.3	Closing Comments	26
\mathbf{A}	Ap	opendix	28
	A.1	Boost library	28
	A.2	Application Programming Interfaces	28
	A.3	Definitions	29

Abstract

This project was an attempt at developing an object detection and tracking system using modern computer vision technology. The project delivers an implemented tracking system. It consists of a hybrid of optical and modern infra-red technology and is applicable to areas such as unsupervised surveillance or semi-autonomous control. It is stable and is applicable as a stand alone system or one that could easily be embedded into an even larger system. The project was implemented in 5 months, and involved research into the area of computer vision and robotic automation. It also involved the inclusion of cutting-edge technology of both the hardware and software kind. The results of the project are expressed in this report, and amount to the application of computer vision techniques in tracking animate objects in both a 2 dimensional and 3 dimensional scene.

All references in the text to "the author" refer to the author of this report.

Acknowledgements

The author would like to thank the following list of people and groups for their technical support and guidance throughout the project:

Dr. Eleni Mangina (Supervisor)

Dr. Mangina acted as the project supervisor. She provided guidance and support throughout the project in the form of project management, specification and review of the report.

Dr. Mauro Dragone (Moderator)

Dr. Dragone was the project mentor and moderator. Dr. Dragone provided support in the form of face-to-face meetings, Skype online discussions, technical advice and report review. Dr. Dragone, along with the help of Dr. Mangina, helped direct the author in determining the scope of the project and handling necessary modifications to the project specification.

Mr. David Swords

Mr. Swords provided the author with a primer on the ROS system during the early stages of the project. His help was invaluable in shaping the author's understanding and confidence in interfacing with the ROS system.

UCD Robotics Group

For allowing the author to participate in the first ever UCD robotics day and showcasing the project to all who attended the event.

Dr. Guénolé Silvestre

For providing insight into correct scientific conduct during projects and general scientific analysis of project work.

1.1 Preamble

This project involved implementing real-time tracking software. This chapter provides a discussion of the project specification. It also gives a high-level overview of the system, leaving design and implementation details for discussion in the respective chapters. It also provides a "roadmap" for the reader about the overall presentation and structure of the report.

1.2 Initial Specification & Overview

The project goal was to produce a working system for tracking objects in 3 dimensional space. The aim of the project was to begin from this spec, and design a solution to the problem. After a satisfactory solution was designed, the task came to implement the solution. Throughout the project, many problems arose. These problems varied from performance issues related to code, and from implementation issues related to limitations of software technologies used. All attempts at overcoming these problems are discussed in this report. Also, as is common case with research, as the project progressed, the resulting tracking system presented in this report differs from the initial design specification; it is more generic and focuses on not tracking preempted objects but any objects that enter the sensor field of view (fov). Chapters 3 and 4 describe the design and implementation of the project and discuss the approach taken to go from specification to the system presented alongside this report.

The system incorporates new technology in an attempt at the tracking problem. The system is driven by data streamed from the XBox $Kinect^{tm}$, and different components of the system process the data in different ways. The center of the system is a user controlled graphical interface for providing feedback and semi-control of the components. A high level description of the system is shown in figure 1.1. The figure describes the system as a series of components each with their own specific duty to perform. The components are all launched simultaneously via the ROS system (more detail on this in chapter 4) and all shutdown when the user signals the GUI to close. The GUI provides the user with up to date tracking information via the 2D optical tracker and the 3D depth based tracker. Both trackers provide visualization capabilities, the optical tracker providing an RGB video stream and the 3D tracker providing a point cloud visualization of the scene.

The next section provides an introduction to tracking, outlining the Kinect sensor and providing some detail of the software technology used in the project.



Figure 1.1: High level illustration of the breakdown of system component delegations. Data is generated from the Kinect sensor, and passed to the GUI and the optical and depth based tracker facilities. The GUI is centered, as it provides a means to communicate with the other components.

1.3 Tracking

As stated, the goal of this project was creating tracking software. This task is an ongoing topic for research, with continous development of algorithms and implementation code. Results of this research have been applied to a range of products such as video surveillance, robotic vision and autonomous flight. In robotics, tracking is frequently used to provide a means for localization and mapping of an unknown environment. One example of this has been the application to enable robots to act human-like and play games like soccer [DOL+05]. The tracking system implemented with this project incorporates the use of the Kinect sensor to provide real-time depth analysis. The sensor is described in more detail in 2.2.1.

Using this data, it is possible to track an object in three dimensions. Applying a mathematical filter to eliminate noise readings from the sensor, a smooth, reliable tracking system is implemented. The filter in question, a Kalman Filter [WB06], is applied in two dimensions (xy plane) to smooth results of the positional data. How this is extended to incorporate the 3D data is explained in chapter 4.

1.4 Justification

There were a number of reasons for undertaking this project. It provided an interesting subject for research, as it combined modern sensor technology with an older problem. The project also implemented a solution that has many unique attributes:

- As the sensor is rather cheap, the project is a cost-effective alternative to other tracking systems.
- The software makes use of open-source implementations. The sensor is interfaced via an open-source library consisting of drivers and an API. At no point is the project

	Discusses reading material, tech-					
	nologies used and related work. It					
2 Background Research	also discusses in detail the algo-					
	rithms that are implemented in the					
	project					
	Design and structure of the project					
3 Design & 4 Implementation	API's and limitations of software					
	used. Details of application code.					
5 Deculta & Conclusions	Performance benchmarking against					
5 Results & Conclusions	a similar software implementation.					
	Critical review of the software.					
6 Future Werlt	Suggestions for improvements and					
o Future work	future work are proposed here.					

Table 1.1: Table charting the structure of the report. All the chapters are presented in the order dictated by this table.

hindered by proprietary technologies.

- The solution uses the Robot Operating System (ROS, discussed in more detail chapter 3), an open-source framework for robots. ROS is developed by an extremely active community and provides high end abstraction of many low level details regarding robotics.
- It provides an interesting baseline to robotic development and computer vision.

The project is also extendable; A desired feature of the software was to provide a control framework for the helicopter along with the tracking module. This can be specified as an extension for a future final year project building on the work here (this is discussed further in chapter 6).

Throughout the project, Object Oriented principles were followed as best as possible. In chapter 3, the project design is discussed in detail, using UML diagrams to show the functional connections between classes. Documentation was also kept, providing a readable, maintainable code base. Finally, many disciplines learned throughout the UCD computer science degree were used to implement the software. This led to a project that demonstrates an ability reachable by a computer science degree.

1.5 Roadmap

This document is structured in a top-down fashion. Chapter 2 discusses reading material that proved useful throughout the project. This is proceeded by core chapters which focus on the details of the project; design, implementation and results & conclusions are all presented and analysed for the reader. Table 1.1 gives an indication to the structure of the report.

The end of the report features an appendix of any elements of the project that may require extra coverage, but are outside the scope of the project.

2.1 Preamble

Building on what has been stated in the previous chapter, this chapter details the reading material that was covered throughout the project. As obstacles were met in the implementation of the project, it was necessary to re-think particular areas of code (algorithms, local optimisations etc.) and change the implementation to allow for better performance among other extensions. Research and reading carried out as the background research related to the project are described. This chapter will outline key papers that described the general computer vision techniques included in the project and reflects on what was studied.

2.2 Technology

As the goal of the project was to create a tracking system, it was important to first understand the state of the art technology being used today, and the algorithms these technologies use. With the main component being the XBox Kinect^{tm} , It was necessary to become familiar with the camera's technology to be able to successfully employ it in the project.

2.2.1 XBox Kinect tm

The Kinect sensor is an RGB-D (color and depth) sensor and operates via twin optical cameras and a LIDAR sensor on the front of the device. Also located on the front of the device is a multi-array microphone system for detecting sounds from different angles. These sensors operate in conjunction with one another to provide a 3D sensor capable of tracking objects and triangualting sound from all angles within a designated surrounding region. Developed promarily for consumer entertainment purposes, its use in active research has become popular due to its cheap price range in comparison with competing devices of a similar capability. [Sun11].

Although an official SDK for the Kinect exists (it was released by $Microsoft^{tm}$ in the summer of 2011), this project is implemented using open-source alternative software. The OpenNI (www.openni.org) API was released shortly after the Kinect sensor, and allows accurate interpretation of the Kinect's data streams. It was developed as part of a joint project to enable the open-source community to develop with this new technology, and the low level drivers are used in this project as a means of interacting with the Kinect.

2.2.2 Robot Operating System

The Robot Operating System (ROS) was used extensively for this project. An open source system for robotics, ROS allows easy development of software used to interact with a range of robotic devices and middleware [QGC⁺09]. The ROS interface for the Kinect was used in this project due to it's message passing framework for processes operating on the same machine and even across other machines. Also the strong, active community provided a platform for help on matters relating to compilation, execution environment and algorithms implemented in the ROS packages. This system is discussed in more detail in chapter 3.

2.2.3 Point Cloud Library

The Point Cloud Library (PCL) was another open-source system used extensively in the project. PCL is a high-level API for 3D rendering and visualization and was used to interface with the Kinect's 3D data output [RC11]. PCL can be used in tandem with ROS, and it was chosen for this purpose as a way to manipulate and understand 3D data from the Kinect. PCL is also covered in chapter 3 where the tracking system developed is discussed in more detail.

2.3 Algorithms and Research

As well as understanding the software and hardware technology used in computer vision, extensive background reading was done on the algorithms and techniques widely used in computer vision applications. This section gives an overview of some of the papers read and provides citations of the papers noted. For some sections that require some extra explaining, the reader is referred to the appropriate appendix section.

2.3.1 Object Detection in 2D: Foreground & Background Segmentation

There are many algorithms for detecting objects in a data stream. One such algorithm that was researched and tested for this project is described in [HLGT03]. Videos containing background objects under various conditions pertaining to lighting and movement were processed and moving objects were segmented from the foreground based on classification under Bayesian rules (See A.3). The background is modelled according to the probability of a pixel being in the background or foreground. This is used in determining feature vectors and classifying by the a prior probability.

Feature vectors chosen for each pixel mainly consist of color composition. Color co-occurance, where a group of neighbouring pixels can share the same color properties, is also incorporated by maintaining the values at time t - 1. This is structured into the feature statistics table shown by [HLGT03], and parameters for the quantization of the image pixels and the size of the statistics table (number of features to consider) were chosen through investigation to yield good results.

The main algorithm in the paper is based on the statistical modelling given, where a number

of features are detected by following the steps of the algorithm. The algorithm can be broken down into four steps (Detailed description is omitted here as it is not relevant; the reader may wish to refer to the citation):

- Pixels of insignificant change are filtered via temporal differencing (See A.3), where analysis of previous state can be used to identify pixels that haven't changed. Pixels are classified by this background and temporal differencing.
- If a pixel has been detected as being temporal, it is classified as a motion pixel and is further classified by feature vector comparison and reference to the feature table as being foreground or background.
- The point set is smoothed by applying a filter operation to any wrongly labelled background pixel to segment the foreground objects.
- The background reference image is updated every time a new point of 'insignificant change' is found [HLGT03].

The paper then goes on to compare the results obtained against benchmarked algorithms mainly relating to gaussian distribution methods. This alogorithm was originally intended for use in the project but was switched after some testing for the algorithm below. The following algorithm provided a less authoritative tracker, but a quicker frame rate.

2.3.2 Object Detection in 2D: Adaptive Background Mixture Model for Real-Time Tracking

The algorithm chosen to implement the 2D tracking system was an implementation of the algorithm proposed by [KB01]. This algorithm is an improvement on [SG99], where the authors implemented a "recent value" set in order to keep the background model up to date. The original algorithm used a non-parametric update with thresholding by maintaining a background model of the scene via a reference image. Background subtraction is processed on the scene to segment the foreground objects. The background model is based on a mixture of gaussian distributions, and different weights are used to represent the temporal portions of colors in the scene. When marking a pixel, weights are applied to each gaussian, and the one with the most probable outcome is used to mark the pixel. As Bowden et al. claim, this method was detrimentally affected by the stochastic update functions used to update the background model. Bowden et al. improved on this by using an expectation-maximisation approach that uses update functions considering the L latest windows of gaussian models.

This algorithm features in the 2D object tracking feature of the project via an OpenCV implementation of a blob tracker. This blob tracker was used to estimate the location of the helicopter in 2D and forward processed output to the 3D tracker.

2.3.3 Kalman Filter

Another important aspect of the project was handling noise output from the sensor. All data output from the Kinect is subject to noise, and for this reason it was necessary to process the data through a filter.

A highly cited filter is the Kalman filter. It is a means to compute, with minimal error, the state in which a process resides [WB06]. It employs mathematical functions to estimate the state of an object (or more abstractly a process) in the past, present and future and it can do this when presented with an unknown model. It is a particle filtering technique, a means of estimating the true path of a model system (in this case an "object") during its movement. Operating on a "discrete-time controlled process" [WB06], it will allow for the filtering of noisy data received by the tracking system of the helicopter. The inputs to the filter are the (noisy) current x-y-z co-ordinates of the helicopter, the orientation and perhaps the velocity/acceleration, while the output is an estimated state. This "state" is representative of the tracked helicopter, and can be used to compute the future state of the system.

The filter has a number of components. It keeps track of its previous state x' in time step k-1, referred to as the a priori state [WB06], and, using a given measurement z, it tries to estimate its current state x'' at step k. With these values come margins for error, and these are represented as linear equations. These errors are known as the a priori and a posteriori estimate errors [WB06]. The error covariance can then be derived as the matrix consisting of the error estimate and its transpose. Finally all these values are used in the following algorithm which implements the Kalman filter (shown here as pseudo-code):

```
While (true) {
    Compute a posteriori state estimate from a priori estimate;
    Update state using this estimate;
    Continue;
}
```

Also, the filter has an equation for representing the difference in distance between the true value and the predicted value called the "residual" [WB06]. If this value is to be zero at any time, this means that both values are in agreement, and the chances of achieving an accurate filtering is high. The equation for computing the new state involves the previous state and linear combination of the resulting difference between observations, weighted to a factor K [WB06]. Welch et al attribute the weighted factor K to describing the "trustworthy-ness" of the estimate. As the error covariance for the measurement approaches zero, the actual measurement received as input to the filtering step can be trusted with being very likely. Conversely, if the a priori error covariance value falls, the real estimate can be regarded as less accurate and the predicted value should be considered as more reliable [WB06].

In its second half, the paper on the introduction to the Kalman filter begins to describe the extended Kalman Filter. The extended version of the filter is used when the measurement process relationship is not linear [WB06]. This project focused on tracking moving objects space. The measurements taken to express their state were measurements of position and orientation, both linear equalities. The Kalman filter features in the optical tracker component of the project (See chapter 4, section 4.3).

3.1 Preamble

This chapter introduces the project formally, providing a specification of the design and discusses the ideas and implementation issues that had to be considered with this design. It outlines reasons for the inclusion of certain technologies and their appropriateness to the task at hand. The chapter also discusses design issues from a software perspective; software methodoligies that were adhered to as much as possible, such as Object Oriented Design principles, Test Driven Development, software documentation and version control software. Finally, it closes with some comments about the design and how the project outcome was affected by design decisions made early on.

3.2 Design Specification

As stated previously, the project specification was to investigate and build a system for robust tracking of an object in 3 dimensional space. More formally, the objective was for a software solution which would enable real-time tracking of this object which would then ultimately lead to the foundations for a fully-automated control system. In order to achieve this goal, the project design had to capture all aspects of the system at run-time, and represent the flow of data through the system. Figure 1.1 in chapter 1 shows a high level outline of the system.

The system was originally designed, from a software perspective, as a single process application that ran in a multi-threaded environment. This meant that all components of the system ran in seperate threads, with the main thread being allocated to the GUI. While this design provided a simple means to implement the system, there were many problems with it.

- This violated the ROS design. ROS is a peer-to-peer system for implementing IPC (inter process communication) (See A.3) [QGC⁺09]. This is to encourage an online system implemented through different processes utilising message passing techniques, perhaps even on different machines. A single process application would oppose this.
- The project was implemented completely in C++, using the Qt framework (See A.2.3). Qt provides its own communication framework known as 'signals and slots'. Problems arose in API (application programming interface) compatabilities. As is discussed in chapter 4, the project was developed using many API's in order to abstract the low-level detail of the sensor and data formats. The OpenCV highgui library (See appendix A.2.1) is used to visualize the optical tracker. This visualization cannot be made using multiple-threads as the API requests that all interactions with window displays be made from the main GUI thread. This was not the case in the original system design.
- A similar problem occured in the 3D component. The point cloud library for visualizing the sensor data requires same thread access to the visualization window. This would not be possible with the original design.

Taking all this into account, it was obvious that a redesign was in order. The revised design is based on IPC between three main executables. The optical tracker component located in *tracker_exe.cpp* implements the 2-dimensional tracking facility. The component is self contained, and utilises ROS to subscribe to and publish RGB image data. The 3D visualizer code, located in *cloud_exe.cpp*, deals with the 3-dimensional data. It subscribes to messages published by the optical tracker in order to process it's own point-cloud data and display it to the user. The third and final component is the GUI contained in *main_window.cpp*. This process subscribes to all published data, providing the user' of the system with real-time feedback of the trackers and allowing them to shutdown the system.

3.3 Software Practices

All application code in the project is coded following an object oriented design. All components in the system are derived from objects, which have their own private attributes and provide an interface in which they can be accessed by other components in the system. This approach provided a means to abstract the system into components easier, as all operations were designed as interactions between objects in the system. It also made sense during implementation, as the classes in the system have a predefined objective and workload. For example, the 2D tracker only needs to interpret a video stream from the sensor and it can then process it internally, with little (or no) interaction with any external component of the system, other than when it goes to publish its results. The fact that there really is no need for shared memory at all also made the design of separate processes more feasible and a natural goal to focus on.

Test driven development was also applied in the project's development. Using the google test framework gtest, test code was written to help validate application code. While this policy was not strictly adhered to, it was followed as best as possible, in some cases only being deviated due to the difficulty in writing tests for message passing code. Also, as the project was not a software development based project rather a research based, the emphasis was on testing and evaluating modern computer vision technology rather than producing industrial standard application code.

An important part of the project's implementation was of course revision control. Throughout development, backups were created using the Git revision control system (See A.3). Git was chosen partly due to familiarity, with this project's author having used it in other projects, and it's simplicity. Git is very powerful, and a complete project repository can be created in just a few commands. Remote backups of code were pushed to the remote repository. This allowed for a safe environment in which code could be written. The branching ability of the revision control system also made it simple to create a new branch for developing a particular part of the system without affecting the current master stable build. When satisfactory, this branch could then be merged into the master build and the process iterated over again. This solution is much cleaner and less prone to error than the typical backup process involving creating archives of files and numbering them for logging.

3.4 Work Breakdown Structure

Final Year Project Nov-Dec Manager: Daniel J. Finnegan

November 1, 2011

Start:

The project design specified 5 months of coding implementation. During these 5 months, application code was written parallel to learning the required API's used in development. A lot of time was spent investigating other solutions, to acquire greater knowledge of the problem domain and how the task has been attempted before. A diary was kept (http://csserver.ucd.ie/~dfinnegan/forth_year_project/index.html) which reported discoveries and logs of data captured with the Kinect. Initially, the project focused on building upon the RoboEarth software [WBC⁺11]. The software for object recognition and tracking is available via the Roboearth stack in ROS. Work began with this software, building models (Figure 4.4) of the helicopter and using these point cloud models to track the helicopter. During this period, it became apparent that the roboearth software was limited to using the accompanying marker template to allow for the object space to be defined where model generation could take place. It was for this reason that the decision to pursue a hybrid tracker was made.

Figure 3.1 shows a gantt chart relating to the work completed in the initial stages of the project. It shows the period spent on the roboearth software and the gathering of core technologies that would be used in implementing the final project. This breakdown of work allowed for contingencies in the project, as time was managed by creating periods of work, where reading and implementation would be done in tandem, saving time and avoiding pitfalls such as deadlines and over-running of reading. The work breakdown structure (WBS) was an important consideration in the design of the project, as it would control the project's flow and also define a strict plan, allowing for a steady and incremental approach to taking the project from specification through to solution and implementation.

	chart								_			
BS	Name				Week 4	5, 2011	Week	46, 2011	Wee	k 47, 2011	Week 4	18, 2011
			1 2 3	4 5 6 7	8 9 1	0 11 12 13	3 14 15 16	17 18 19 2	20 21 22 2	3 24 25 26 27 2	8 29 30	1 2 3 4 5
	Study and learn API (R	DS)										
	Write application code	to test ROS										
	Background reading											
	Development environm	ent setup and co	n									
	Investigate Roboearth :	software										
	Investigate OpenCV											
	11)	•									
ks												
BS	Name						Start	Finish	Work	Complete	Cost	Assigned
	Study and learn API (ROS)					Nov 1	Nov 18	14d	100%		
	Write application cod	e to test ROS					Nov 4	Nov 17	10d	100%		
	Background reading						Nov 1	Nov 28	20d	100%		
	Development environ	ment setup and o	ore techno	ologies chos	en		Nov 24	Nov 25	2d	100%		
	Investigate Roboearth	n software		-			Dec 1	Dec 20	14d	100%		
	Investigate OpenCV						Dec 15	Dec 28	10d	100%		
soui	ces											
ame	Sho	t name			Type		Grou	D		Email		Cost

Figure 3.1: A gantt chart representing the work done during the initial stages of the project. At this point, the final version of the software had not been defined yet, and the main background research and environment setup were engaged.

The next chapter builds upon the discussions here and gives more detail on the class design of the system and the components which make up the tracking software.

4.1 Preamble

Following on from the previous chapter's introduction to the project design, the goal of this chapter is to inform the reader of the actual implementation of the code. It details the steps taken to render 3D point clouds from the Kinect, how the optical tracking component works, and the overall integration of all the components that make up the system. Brief analysis of the project's code is made and discussed, along with the technologies and API's used in implementing the project. Unified Modeling Language (UML) diagrams are presented which illustrate the system and the interactions between components. References are made to parts of the code which perform key operations in the system.

4.2 System Overview

The system is comprised of 3 main components; the optical 2D tracker, the 3D visualizer, and the GUI. A typical use case of the system is presented in figure 4.3. The system is initiated via the roslaunch command. This command reads the launch configuration from the systems launch configuration file and boots all the components required for the system to run. Although manual initiation is possible by simply executing the components seperately, this configuration provides a clean shutdown using ROS's shutdown hook.

During execution, the system process' two main data streams published by the sensor. The system interprets the RGB image stream using the optical tracker and the depth points stream using the visualizer. Some components also generate and publish their own messages. Any interaction with the system is done via the GUI. The interface is composed of a single text panel which is updated dynamically every time a new object location is found. The interface also has a button for shutting down the the entire system. Figure 4.2 shows a class diagram interaction for the nodes in the system.

Most of the implementation of the project occured during the post-christmas period, throughout the holiday period. This gave ample time to collect log data that could be used in testing the system, as well as providing time to use the roboearth software, build models and gather results and further logs. Figure 4.1 shows a gantt chart of the weeks leading up to the project completion.

The following sections are dedicated to the individual components of the system. Each section breaks down into subsections to discuss in finer detail parts of the component that deserve separate attention.

Final Year Project Feb-March

Manager:	Daniel J. Finnegan
Start:	February 1, 2012
Finish:	March 20, 2012
Report Date:	March 5, 2012

Gantt (Chart								
WBS	Name	Work		Week 6, 2012	Week 7,	2012 Wee	< 8, 2012 I	Week 9, 2012 V	Week 10
			1 2 3 4 5	6 7 8 9 10 11	12 13 14 15 16	17 18 19 20 21 22	23 24 25 26 27 2	28 29 1 2 3 4 5	6 7 8
1	Tracker executable	7d							
2	Ongoing testing	35d							
3	Visualization executable	7d							
4	GUI	16d							
			4		11.				Þ
Tasks									
WBS	Name		Start	Finish	Work	Complete	Cost	Assigned to	
1	Tracker executable		Feb 1	Feb 9	7d	100%			
2	Ongoing testing		Feb 1	Mar 20	35d	50%			
3	Visualization executable	5	Feb 8	Feb 16	7d	90%			
4	GUI		Feb 15	5 Mar 7	16d	80%			
Resour	ces								
Name	Short nar	ne		Туре	Grou	р	Email	Cost	
								This file was generated	by Planner

Figure 4.1: A Gantt chart of the refactoring code during February and March 2012. At this point, development focused on improving the code quality and completing components such as the GUI. The chart shows onging testing, that was specified for an arbitrary date after project completion to emphasize that the testing would continue throughout the entire project.

4.3 2D Tracking Using OpenCV

The 2D optical tracker is implemented using the computer vision library OpenCV. OpenCV is an extensive vision library implemented in C++ and Python, with other language wrappers. The library provides implementations for many different computer vision algorithms, creating an API that is rather simple to use and that can yield powerful results with little code. OpenCV plays a part in this project as the basis of the optical tracker component for the system. The system makes use of OpenCV's Kalman filter implementation and of the algorithm described previously in 2.3.2 and cited in [KB01]. The following subsections detail the optical tracking system.



Figure 4.2: Class diagram showing the interaction between the tracker and visualization components with each other and with the sensor. The notation specifying the none to many relationship is true, as any number of nodes can subscribe to a certain topic in ROS. However in this system there is only one of each



Figure 4.3: Use case diagram showing simple interaction with the system, in this case the user initialising the system via roslaunch

4.3.1 Object Detection and Blob Tracking

The tracking is implemented using the Blob structure of the OpenCV auxillary library *cvaux.h.* The *tracker* class in *tracker.hpp* creates instances of these blobTracker modules. It acts as an accessor around the module, allowing for static initiation of the tracker utilising the algorithm chosen and it contains methods for processing images by passing them through the tracker pipeline. The pipeline involves subtracting the background via the mixture of gaussians algorithm, then detecting moving objects and assigning a "blob" ID to each object and tracking it through space. One of the main criticisms the author has towards the API is the lack of documentation of the code. The OpenCV site http://opencv.willowgarage.com/wiki/VideoSurveillance acknowledges this defect in the code and provides a "best possible" documentation of what the code does at a high level.

In order to utilise the tracker in the project, it was necessary to look at the source code of the samples provided with the OpenCV library. While also not very well documented, the pipeline can be setup relatively simply with a few lines of code. This initiation protocol is wrapped by the *tracker* class in the *setup_tracker()* method. After having instanciated the blob tracker, the optical tracker component can then subscribe to the image topic published by the sensor and enter the main ROS event loop. It is in this loop that all callbacks are handled. To allow for a steady frame rate, the tracker operates at 20 fps using a buffer capped at 10 messages. ¹

The final piece necessary was the use of the ROS cv_bridge stack. This stack is used to convert the ROS messages to the format required by OpenCV, as the tracking implementation uses the old IpIImage format originally used in the intel image processing library. A considerable source of difficulty during the project was in the various conversion utilities required between different data formats. As the project uses a mix of cutting edge software and this outdated format, a lot of time was spent on remedying compiler issues and linkage errors by using deprecated versions of the API's.

4.3.2 Blob Recording and Message Passing

After processing the image and tracking the blobs, the optical tracker component must then send this data to the 3D visualizer. This is implemented by creating custom messages via the *rosmsg* utility. Two custom messages were created for the tracker component (defined in the *Blob.msg* and *VectorBlob.msg* files). After gathering the co-ordinates of each blob, this data is packaged into a Blob object. At one cycle of development, these blobs were packed into a vector, allowing a single VectorBlob message to be published. After testing and further design analysis, it was decided to just publish the location of the largest blob in the scene, thus making use of just the *Blob.msg* file.

The custom messages were generated into valid C++ classes using the *rosmsg* utility. This is the standard method for implementing custom messages in ROS. All messages are passed as shared pointers using the boost C++ library for safe memory management (See A.1).

¹This was influenced by the OpenCV highgui libraries framework for displaying video images. In order to render the images, the main event thread must be paused with a call to cvWaitKey(). When this value was set to a high delay, the sensor would be publishing messages far too quickly for the tracker to maintain a steady frame rate.

4.4 3D Visualization and Augmented Tracking With PCL

In order to render the depth point cloud data published by the sensor, the Point Cloud Library (from here on referred to as PCL) was used (See A.2.2). PCL is an industry standard library for processing 3 dimensional point cloud data. It was chosen as the 3D rendering engine for the project due to its integration with ROS. As in the optical tracker implementation, there were significant problems in using the API due to changes in the latest stable release and ROS not being up to date at the time of this project's implementation. PCL features many high level implementations of general point cloud processing applications, and provides class based implementations of algorithms related to 3D processing. PCL's high level visualization toolkit, discussed below, was used to incorporate 3D data from the Kinect into the project. The following subsections aim to describe further the use of PCL in the project to render 3D point cloud data.

4.4.1 Visualizing Point Cloud Data

At the heart of the visualizers implementation is PCL's PCLVisualizer class. This class is a complete rendering toolkit, implementing an OpenGL-rendered scene of the input 3D point cloud. The visualizer also has keyboard hooks to respond to user input, and operates via it's own internal event loop. The visualizer in this project wraps around this class by utilising the live stream from the kinect to update the cloud rendered. Point clouds are passed to the visualizer through ROS and then displayed in the viewer.

The aim of the 3D point cloud was to provide an augmented tracker; one that would track the object in xyz dimensions allowing for a global estimate of the query object. As the roboearth software applied a point cloud model of the object in question, this goal of this project was to first attempt at recreating this behaviour and then improving the tracking system. Using the software, a model of a helicopter was built and this model was used as input to the system. Figure 4.4 shows a screenshot of the model being successfully located in the cloud.

Although the RoboEarth software utilises a point cloud model of the object being tracked, this project does not focus on robust model based tracking but instead is an attempt to create a hybrid tracker by combining the 2D and 3D data from the sensor. In order to do this, it was necessary to build the 3D visualizer so that it would subscribe to data topics from the sensor itself and from data published by the 2D tracking component. Therefore, before a cloud is visualized, it must be processed first. This processing is lightweight, involving an indexing routine into the cloud data in order to extract key points. The code relating to this can be found in *cloud_exe.cpp*.

4.4.2 Augmented 3D Tracking

Normally, it would not be possible to use the output of the 2D tracker directly with the 3D visualizer. However, the OpenNI drivers for interfacing with the Kinecttm sensor allow for a depth registered point cloud to be output. This cloud is a processed cloud consisting of the optical camera's data and the IR sensor's data in order to output a cloud featuring a 1-1 mapping for the RGB image and point cloud data. This made it possible for the system to simply query the raw point cloud data. As the blob tracker utilised by the 2D tracking component outputs pixel co-ordinates of a tracked blob in the scene, these values can be used



Figure 4.4: A screenshot of the roboearth software being executed with a model built using the software. The screenshot shows the helicopter being detected, shown as a red highlight in the software rendering window.



Figure 4.5: A screenshot of the tracking system tracking a small RC Lamborghini car. The image shows the red marker box where the tracking system has estimated the car to be.

to index the point cloud's raw data array. Upon locating the query point, the xyz co-ordinates of the point in the cloud (relative to the sensor device) can easily be read from the point itself.

4.5 Message Passing in ROS

In order to process the point clouds mentioned in 4.4.1, the 3D tracker partitions the work into two threads. The main thread of execution focuses on handling the ROS integration, implementing the required callback methods to gather the data published by the sensor and by the optical tracker. These callback methods simply buffer the data into two queues to be processed by the second thread. Both threads implement a simple version of the producer/consumer problem through the use of boost mutexes and lock mechanisms (for details regarding the boost library, see A.1). The threads feed into and consume from a pair of STL (See A.3) queues holding blobs and point clouds. Whenever the consuming thread sees that neither of the queues are full, it pops the head of each queue and uses the popped blob and point cloud in order to create the annotated cloud, showing the estimated location of the blob and then passes this to the underlying cloud viewer for visualization to the user.

This P/C implementation is not regarded as the safest or most thorough implementation possible. Likely improvements that could be made to the code (as well as any related optimisations) are discussed in chapter 6.

4.6 Summary

This chapter presented the system to the user from an implementation perspective. It discussed the components of the system and their role in the systems execution. Details on implementation issues regarding external libraries were given, with references to exact uses in the system. An indication to how the system operates was provided via the use of UML diagrams and class interaction illustrations. All code snippets were withdrawn, opting for documentation references in place of cluttering code functions. In the next chapter, the results of the program are presented for critical anlaysis. All criticism is from the authors own viewpoint, with comparisons being made between the system presented in this project and with a benchmark system provided as part of the RoboEarth european funded project on robotic advancement [WBC⁺11].

5.1 Preamble

This chapter focuses on presenting the results of testing the run-time efficiency of the application in comparison to the RoboEarth software. As both systems use different methods for tracking, it would be unfair to compare the systems based solely on the tracking results. Although such a comparison is discussed to support the performance of this project's system, a more fair and unbiased comparison of system resource consumption is made. This comparison is based on CPU and memory usage of both systems during execution. In the final subsections a recap of the work made is used to establish conclusions of the project, lessons learned and the author's own critical review of parts of the project.

5.2 Performance Benchmarking

This section provides analysis of the results obtained in comparing the performance of the system against the roboearth tracker. The reader is informed that during the performance testing, both systems were running on an intel i3 core processor, clocked at 2.4GHZ and running the Ubuntu operating system. All benchmarks were recorded while there were no user processes running other than the tracking systems. The reader is also reminded what these tests show. They do not compare the proficiency or accuracy of the results (though this comparison is made briefly in section 5.4). Instead they show the resource usage of both systems, as one of the author's main concerns was the speed of the roboearth tracking system. It was decided that a valid justification of the system presented here would be how it performs relative to this.

During the benchmark testing, the roboearth system was ran against a model of a small radio controlled helicopter. This model was built with the roboearth model creation software (distributed with the roboearth ROS stack). The tracking system reported in this project was ran against a radio controlled race car. As is visible from figures 5.1 and 5.2, the tracker system surpasses the roboearth system in terms of CPU usage, with the roboearth system regularly freezing during execution and showing a high I/O wait time. The reason for this is the high memory requirements of the roboearth system. The software requires that a model of the object being detected be held in memory, so as every point cloud rendered via the sensor data can be compared against the model in order to extract it's location. The blob tracking system has no such requirement, and is therefore much more lightweight.

In order to elaborate this further, a memory usage comparison was also graphed. Figures 5.3 and 5.4 show the percentage of memory used throughout execution of both systems. The graph shows the expected result based on the previous CPU benchmark; that the roboearth system's model residing in memory is the biggest factor in the memory consumption.



Figure 5.1: Graph representing CPU usage during execution of the Roboearth object detection software. The graph shows a high I/O wait time, as the processor was busy accessing memory and awaiting input due to the large data of the helicopter's 3D point cloud model. The graph measures CPU usage against system timestamps. The blue line represents the user processes running, in this case the roboearth software, and the black line shows the I/O wait of the processor.



Figure 5.2: Graph representing CPU usage during execution of the tracking system. In contrast to the roboearth system, the tracker shows more optimal CPU usage, as without the bulky model, less time is spent on I/O waits while accessing memory. The blue line represents the tracker system, comprised of three running processes.



Figure 5.3: Graph representing memory usage of the roboearth system during execution. The blue line shows the application bringing the machine's available memory down below 50MB.



Figure 5.4: Graph representing memory usage during execution of the tracking system. The blue line shows the memory still available for the machine. Although it is still quite low, it is indeed higher than that of the roboearth system.

	RoboEarth	Tracker		
Memory	High memory consumption due to	Low memory consump-		
	object model. Uses 2 sperate pro-	tion, uses 3 processes		
	cesses to perform task.			
Model	Bases it's detection on point cloud	No pre-determined ob-		
	to model comparison.	ject model		
CPU	Low CPU utilisation due to high	High CPU utilisation.		
	I/O wait. Occasionally freezes on	As far as was tested, the		
	the machine the software was tested	system will run indefi-		
		nitely.		
Number of detectable objects	Limited to a single loaded object	Can detect any change		
	model.	in the scene. Blob		
		tracker surpassed		
		50 blob ID's during		
		testing.		

Table 5.1: Summary of the comparison results between the roboearth software and the project's tracking system.

5.3 Critical Analysis

An important caveat to remember in these tests is the accuracy/memory tradeoff of both systems. While these results may seem to make the tracker appear to be a far superior system, the reason it's footprint is so small is due to the lack of a concrete model. The tracker system reacts to movement in the scene. It has no correlation to any model and can therefore not be used to locate a *specific* object in the scene, just report the location of the *largest animated* object detectable in the scene. It is true to say that the roboearth software can detect inanimate objects in the scene, and it does this with a degree of accuracy proportional to the quality of the model used. The roboearth system can also estimate the pose of any object it detects. However, a mutual hindrance to both systems is the problem of occlusion. Both systems are susceptible to a loss in detection if an object becomes occluded by another foreground object. The roboearth system tackles this problem by partial model estimation, the tracking system in this project is assisted by the Kalman filter applied to the main tracking algorithm. Table 5.1 gives a summary of the pros and cons of both systems.

5.4 Conclusions

Conclusions drawn from these benchmark tests clearly show that the hybrid tracker performs better in terms of system resource consumption. This is a very desirable feature, one that could make it possible to execute the system in a constrained environment (such as an embedded sensor environment). Also, the tracker is completely modular; all components that make up the system can operate alone, providing their own processed output from their respective input streams. However, the tracker's biggest (and depending on the application, devastatingly fatal) flaw is it's accuracy. As has been stated countless times already, this tracker implements a segmentation algorithm that allows it to respond to changes in the foreground of the image. In a critical system, where it is absolutely essential to have an accuracy threshold in the region of 90% +, this tracker would not be suitable. However, the tradeoff is that the tracker operates in a less resource hungry manner. The roboearth software is also not to be undermined. The software would surely operate better on a stronger machine, such as a high end desktop machine. Also, another way to improve performance would be to implement the software in a more distributed manner, benefiting from ROS's message passing framework. This optimised distributed framework is described in 6.2.

This chapter suggests possible additions and augmentations to the tracking system which could allow it to operate as part of a bigger system. It briefly states possible improvements that could be made and acts as a closing comment to the report by providing the authors own view of the project.

6.1 Autonomous Control of a Robot

One of the goals suggested at the beginning of the project was to build a helicopter robot, complete with a system control that could be operated by a user or could be implemented as an autonomous control system. One of the modules that would be required for such a task would be a tracker, capable of relaying back to the control center the current location of the robot at any given time. Acknowledging the criticisms given in section 5.3, this tracker could be improved further by perhaps providing a more sophisticated algorithm (such as [HLGT03]) and deploying the system on a faster machine. Then this tracker could possibly be used in such an environment.

6.2 **Optimisations and Distribution**

There are many optimisations that could be made to the code. For example, the threaded nature of the visualizer can cause a delay in the point cloud rendering. This is probably due to the efficiency of the application code. With more time, this could perhaps have been optimised to allow for a smoother rendering process. Another more UI friendly feature that could have been added was a command panel allowing the user to open and close particular parts of the system at will. This however, was not implemented for this project as the author believed it to be unnecessary in demonstrating what the project does.

The most important optimisation as far as this author is concerned would be in creating a distributed environment for the code to run. As ROS natively accomodates the use of multiple machine control of the operating system, it is obvious that the next step would be in utilising this to speed up the system. As the tracker is already modular, components could easily be executed on separate systems, allowing for more processing to be done per frame. This processing could range from adding an extra filtering mechanism to the optical tracker, rendering a faster point cloud, and producing more relevant data than the location of the object (velocity etc.). Of course by lessons learned from the roboearth detector, the system could be modified to also include a model of the object being detected. A single machine could be used to load the model alone, allowing another machine to process the system itself, providing even better performance. This would enable the tracker to filter out unwanted objects and also provide a fast tracking paradigm with pose estimation thanks to the model.

6.3 Closing Comments

This project applied modern sensor technology to the tracking problem. With the help of rich API's, a system that does indeed perform as is stated in this report was built. This report detailed the process in taking the project from formal specification through to design and implementation. The strengths and flaws of the system were discussed and analysed, and a comparison was made with similar system.

This report has hopefully presented to the user with a best understanding possible of the tracking system developed. Accompanying the report is the project implementation along with a generated documentation directory. This directory contains instructions for installation of the system as well as details on the code implementation. It is the author's wish that the reader may attempt to install the system where possible, to get a direct interpretation of the system during execution.

Bibliography

- [DOL^{+05]} Mauro Dragone, Ruadhan O'Donoghue, John J. Leonard, Gregory O'Hare, Brian Duffy, Andrew Patrikalakis, and Jacques Leederkerken. Robot Soccer Anywhere: Achieving Persistent Autonomous navigation, Mapping and Object Vision Tracking in Dynamic Environments. SPIE Opto Ireland, 2005.
- [HLGT03] Weiman Huang, Liyuan Li, Irene Y.H Gu, and Qi Tian. Foreground Object Detection from Videos Containing Complex Background. 2003.
- [KB01] Pakorn KaewTraKulPong and Richard Bowden. An Imporoved Adaptive Background Mixture Model for Real-Time Tracking with Shadow Detection. 2001.
- [QGC⁺09] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: An open-source Robot Operating System. In Proceedings of the International Conference on Advanced Robotics (ICAR), Munich, Germany, June 22-26 2009.
- [RC11] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In IEEE International Conference on Robotics and Automation (ICRA), Shanghai, China, May 9-13 2011.
- [SG99] Chris Stauffeur and W.E.L Grimson. Adaptive Background Mixture Models for Real-Time Tracking, 1999.
- [Sun11] Kelvin Sung. Recent Videogame Console Technologies. *IEEE*, 2011.
- [WB06] Greg Welch and Gary Bishop. An Introduction to the Kalman Filter, 2006.
- [WBC⁺11] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Galvez-Lopez, K. Haussermann, R. Janssen, J.M.M. Montiel, A. Perzylo, B. Schiessle, M. Tenorth, O. Zweigle, and R. van de Molengraft. Roboearth. *Robotics Automation Magazine, IEEE*, 18(2):69 –82, june 2011.

Appendix A: Appendix

Listed below are various extensions to references in the text. They are here to provide the reader with extra detail that may be required but was not present in the main report chapters.

A.1 Boost library

The boost C++ library is a library consisting of header files which implement various useful extensions to the C++ language. Boost was used in this project to provide its implementation of smart pointers. A smart pointer is a handle on a region of memory that provides management of the object being referenced. One of its widely used cases is in automatic memory management. With a smart pointer, the object being referenced will be automatically deleted when the pointer itself is destroyed or at another time, dictated by the underlying deletion policy of the pointer. The following code snippet can be used to create one variant of the smart pointer, the scoped pointer:

```
void function()
{
     {
            f
            boost::scoped_ptr<my_object> objPtr(new my_object());
            // Perform some action
            objPtr->get_result();
     }
      // From here on, objPtr cannot be used but there is no
      // fear of a memory leak as the pointer only lives within
      // the above block's scope
}
```

As both ROS and PCL use the boost library heavily for this feature, to comply with this convention (and to also benefit from it) this project incorporates the boost library into its implementation as well.

A.2 Application Programming Interfaces

A.2.1 OpenCV

The project utilised many different API's. OpenCV, the main computer vision API, was originally developed at Intel corporations and is now maintained at Willow Garage (http://www.willowgarage.com) and provides implementations of many widely used computer vision algorithms. The OpenCV project has been in development since 1999, and is

regarded as one of the best high end computer vision libraries available.

A.2.2 PCL

Point Cloud Library (PCL), another branch from willow garage, is an ever-increasing library for 3D processing. Similarly to OpenCV, it provides high level solutions to common 3D processing problems. It can be downloaded from http://www.pointclouds.org and is also freely available and open sourced.

A.2.3 Qt

Qt is an API developed by the mobile phone company Nokia. Originally developed by Trolltech, Qt provides a C++ framework for creating applications which use the qmake build tool. This tool allows Qt to use some of its custom utilities such as the 'signals and slots' framework. Qt was also the main API for development of mobile applications for Nokia's Symbian operating system. For more information see http://qt.nokia.com/products/

A.3 Definitions

Various definitions of terms used in the text. All definitions are given in the user's own words, hence the lack of citations.

Bayesian Rules:

A set of hypotheses that state how a belief or understanding of a system of expressions should change in order to correctly describe the outcome of events, subject to changes in observation.

Temporal Differencing:

A process which filters information based on the time observed.

Inter Process Communication:

A means of sharing data and communicating between processes residing in their own address space within an operating system.

Revision Control:

Automatic file system configuration management. Allows the creation of repositories in a file system for file backups.

STL:

Acronym for standard template library, the standard library for C++ programming. It forms a collection of algorithmic and data structure implementations that are common in everyday C++ programs.