DEVELOPMENT OF COMPUTER VISION ALGORITHMS USING J2ME FOR MOBILE PHONE APPLICATIONS

A thesis submitted in partial fulfilment

of the requirements for the Degree

of

Master of Science

in the

University of Canterbury

by

Jian Gu

SUPERVISOR: Dr. R Mukundan

Co-SUPERVISOR: Prof. Mark Billinghurst (HIT Lab NZ)

Department of Computer Science and Software Engineering

University of Canterbury

November 2008

ABSTRACT

DEVELOPMENT OF COMPUTER VISION ALGORITHMS USING J2ME FOR MOBILE PHONE APPLICATIONS

Jian Gu

This thesis describes research on the use of Java to develop cross-platform computer vision applications for mobile phones with integrated cameras. The particular area of research that we are interested in is Mobile Augmented Reality (AR). Currently there is no computer vision library which can be used for mobile Augmented Reality using the J2ME platform. This thesis introduces the structure of our J2ME computer vision library and describes the implementation of algorithms in our library. We also present several sample applications on J2ME enabled mobile phones and report on experiments conducted to evaluate the compatibility, portability and efficiency of the implemented algorithms.

ACKNOWLEDGMENTS

This thesis would not have been possible without the help of many people. I want to thank foremost my supervisor and co-supervisor Dr R Mukundan and Prof. Mark Billinghurst for their constant support and guidance during the development of this work. They suggested new ideas for me to explore and have helped me appreciate the principles of scientific work. I also would like to thank Anders Henrysson and Charles Han for generously allowing me to use their existing applications in my experiments.

Many thanks to Daniel Wagner and Michael Rohs who provided the source code of STBtracker and Visual code for me to study. I also want to thank all the members of HITLabNZ, which offers an extremely supportive environment for people to carry out research and to explore new research directions. Also, thanks to Adam Chang who had provided some new ideas for evaluation and helping in the checking grammar mistakes in this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

J2ME	Java 2 Micro Edition
BREW	Binary Runtime for Wireless
ExEn	Execution Environment
PVL	Phone Vision Library
AR	Augmented Reality
RAM	Random Access Memory
ROM	Read-only Memory
STBtracker	Studierstube Tracker
FPL	Fixed Point Library
FS	Full Search
TSS	Three Step Search
NVL	Nokia Vision Library

1 Introduction

This chapter gives a general overview of the research project including the primary motivation, the important components and the organization of this thesis.

1.1 Motivation for Research

Mobile phones are becoming an important part of our everyday life. The current standard mobile phones have full colour displays, integrated cameras, fast processors and even dedicated 3D graphics chips. According to Gartner [42], the global sales of mobile phones will reach 1.28 billion by the end of 2008. This figure is a growth of 11% from 1.15 billion phones sold back in 2007. In the Asia Pacific region along, sales are predicted to reach 472.5 million mobile phones by the end of 2008, an increase of 17.9% from 2007.

Camera phones are extremely popular and widespread among mobile phone users. According to Gartner [42], the camera phone market is growing exponentially with worldwide annual shipments up more than 200% since 2005. It has been estimated that by the end of the 2009, approximately 70% of all mobile phones will be equipped with digital cameras. In addition to taking still images, camera phones are also capable of video streaming, using either the camera on the back for recording or the camera in the

front for video phone calls. The tight coupling of camera and CPU gives mobile phones unique input capabilities where real-time computer vision can be used to enable new interaction techniques and link both the physical and virtual worlds. As image resolution improves and computing power increases, camera phones can do more interesting things than just taking pictures and sending them as multimedia messages over the mobile phone network.

However, although mobile phones with cameras are becoming common, there are currently very few phone applications that use camera input for anything other than capturing still images or video. Thus there is an untapped opportunity to develop innovative interactive applications that use mobile phone camera input. This is because unlike standard digital cameras, mobile phones are programmable by third party developers and so developers have access to communications, computing and image capture capabilities in a programmable mobile form factor.

1.2 Computer Vision Applications on Mobile Phones

There are several classes of applications that could be considered. In this thesis we consider three types of computer vision application: (1) General image processing (such as edge detection), (2) 2D image processing (such as motion estimation) and (3) 3D camera pose determination (such as used in Augmented Reality applications). The image processing required for each of these types is progressively more complicated.

(1) General image processing

In a general image processing application there is usually no need for real time performance, and image processing accuracy is more important than speed. To verify the Java static image processing application we will need to write a Java program that will use the phone camera to capture an image, perform simple image processing on it, and then display the result. Once complete, we will know the camera phone's ability to support Java based static image processing. As will be shown later in the thesis there are several examples of J2ME computer vision applications for mobile phones that use static image processing.

(2) 2D image processing

With 2D image processing, simple image processing algorithms are developed that perform 2D image processing on camera images in real time and enable a number of interesting applications. For example, simple 2D motion estimation techniques can be used to detect phone motion. In this case, the user's physical movement of the device is captured in the incoming video stream, which is analysed using image processing algorithms.

(3) 3D camera pose determination

The most complex image processing we are interested in is 3D camera pose determination from a visual marker. In this case by combining the result from 2D image processing with 3D computer vision techniques and a Java 3D game engine we could create an Augmented Reality (AR) [47] application. AR applications are those that

involve the real time overlay of 3D virtual images on video of the real world. This process would involve using the JSR 184 library [36] to render a number of 3D models, and align the virtual objects with real markers in the video frame. The challenge with 3D camera pose determination is that the computational speed of current mobile phones is slow, and so could cause some tracking problems. This could be improved by increasing the processing speed of the Java program by code optimisation.

1.3 Why use J2ME ?

Unlike the development environment for standard desktop computers, mobile phones use a diverse range of operating systems. Mobile applications are developed using platforms and technologies such as Windows Mobile [39], Palm OS [40], Symbian OS [41], Macromedia's Flash Lite [42], DoCoMo's DoJa [43], Sun's J2ME (Java 2 Micro Edition)[44] , Qualcomm's BREW (Binary Runtime for Wireless) [45] or Infusio's ExEn (Execution Environment) [46]. It is generally difficult to develop mobile applications that can run across a wide range of phone handsets without a significant porting effort. However Java and J2ME are designed to provide a platform independent development tool.

Developing a computer vision application that works in real time on an embedded device such as a mobile phone requires a careful selection of techniques. The mobile platform imposes constraints on both computational power and storage capacity. J2ME is one of the most common platforms for mobile applications and games. It has a software library that provides high-level user interface features such as lists and

checkboxes. It also allows a developer to display any content on the screen, and to assign functions to handset keys; this characteristic enables developers to create better, richer, and easier-to-use interfaces for their applications. Applications developed under J2ME are portable between different operating systems. Furthermore, compared to applications created under other platforms, J2ME applications take less time during re-implementation and for porting.

1.4 Organization of Thesis

This thesis explores the feasibility of using J2ME for a mobile computer vision library and describes several mobile applications based on this library. In Chapter 2, we will review related work on camera phone applications. Section 3 gives an overview of our library structure. In Chapter 4, the details of the algorithms used in building mobile augmented reality applications and other computer vision applications will be introduced. In Chapter 5, experimental results are presented. Finally, in Chapter 6, we will draw a conclusion for our research and give some suggestions on how the application could be improved in the future.

2 Background Study and Related Work

In this chapter we give an overview of existing computer vision applications and libraries available for mobile phones. Although there are many games and applications available for mobile phones, only a small proportion of them use camera input. In this chapter, we first describe the existing vision applications and libraries, which are not developed based on the J2ME platform, then we review the existing J2ME based computer vision applications.

2.1 Existing Non-J2ME Computer Vision Applications

We group these non-J2ME computer vision applications into the three types of applications described previously (General, 2D image processing, 3D pose calculations).

2.1.1 General image processing

In a general image processing application, a picture is taken and then processed over several seconds to yield a result. Using camera phones to identify visual tags is one of the most common general image processing applications available. They typically allow a user to take a picture of a 2D barcode pattern which is then processed to load a web page or provide similar information about the tagged object. Three camera-phone based tag

reading packages are commonly employed: QR code [11], SpotCode [12] and Sema-Code [13].

QR code

QR code [11] is based on a two-dimensional square pattern developed by Denso Wave Incorporated (a division of Denso Corporation, Japan) with the primary aim of generating a symbol that is easily interpreted by scanner equipment. Most camera phones in Japan have this software already installed, so users can easily work with QR codes to read web addresses, or encode personal information on a name card or stamp.

Figure 2-1: QR Coded objects

SpotCodes

SpotCodes [12], also known as ShotCodes, are a commercialized derivative of TRIP codes [15]. They identify the rotation of code tags in an image and allow a Bluetooth-enabled camera phone to control active displays as a sophisticated pointing-device

regardless of shape and size of computers. However, they do not provide an orientation-independent marker coordinate system, and do not detect relative camera movement. A number of interaction possibilities are used with a typical web page, such as using rotation controls and sliders.

Figure 2-2 The SpotCode reader performing real-time video image processing

Sema-Code,

Sema-Code's technology [13] allows camera phones to read barcodes and automatically access web pages based on them. The Sema-Code uses standard Data Matrix [16] codes to implement physical hyperlinks, and load web pages in the phone's browser. This can be used to link live information to Sema-Codes on business cards or conference badges, such as the position of GPS-equipped busses, or information on nearby shops and services.

Figure 2-3 Sema-codes

Eleanor [17] evaluated each of these packages in terms of decoding latency, robustness to variable lighting conditions and robustness to variability in the phone's orientation with respect to visual tags. In each of these areas, the ShortCode Reader performed the best of the three.

Other 2D marker technologies include CyberCode [14] and TRIP tags [15]. CyberCodes and TRIP tags are not recommended in mobile phone class devices. This is because CyberCodes can encode only 224 unique tags and TRIP tags just 39 possible values, making it difficult to store the widget type information in the code itself.

In addition to 2D marker tags there are some interesting possibilities for applications that involve non-real time image capture and manipulation. Some of these are listed below:

PullFace

PullFace [18] is an application that allows a user to modify pictures on mobile phones in an elastic fashion. Once a user has taken a picture with his or her mobile phone they are free to modify any part of the image, and can move back and forth through the history of the modifications (undo and redo). Due to this flexibility, users can keep on improving their skills, and find new and funnier ways to play with different images.

Figure 2-4 Pullface Examples

2.1.2 2D Real Time Image Processing

There are a number of applications that use simple image processing techniques to provide real time 2D image processing. These techniques include pattern recognition, edge detection, motion flow and feature tracking among others.

Rohs' Tag Recognizer

Some of the same tag recognition techniques described above can be implemented in real time. For example, Rohs' tag recognizer [19] is a system that turns camera phones into mobile sensors for 2-dimensional visual codes. The software creates a marker coordinate system, and visually detects phone movements, such as rotation angle and the amount of tilting of the camera as additional input parameters. These features enable applications such as item selection and interaction with large-scale displays.

Recently, the system was improved to make an AR game for mobile phones --- Penalty Kick. Penalty Kick [20] uses a coarsely registered 3D marker, which can be printed on a poster or product package. The aim of the game is to shoot a soccer ball into a goal printed on the product package. The player can aim where to shoot the ball by rotating and tilting the phone. The virtual goal keeper will then try to catch the ball.

Figure 2-5 Michael Roh's Penalty Kick

There are also a range of games and other applications that use more complex 2D image processing. Two of the best known are “Mosquito Hunt” [21] and “Marble Revolution” [22]. In these games, 2D motion flow techniques are used to track phone orientation and provide tilt input. For example, in the Mosquito Hunt game the user rotates his phone to target virtual mosquitoes that appear overlaid on live video of the real world. Another variation of this is Hakkarainen’s “Symball” game [23]. This is a two person collaborative table tennis game which uses Bluetooth enabled camera phones and computer vision based motion tracking. The phone display shows the virtual table, net, ball and rackets, and players make shots by tilting the phone to move their racket.

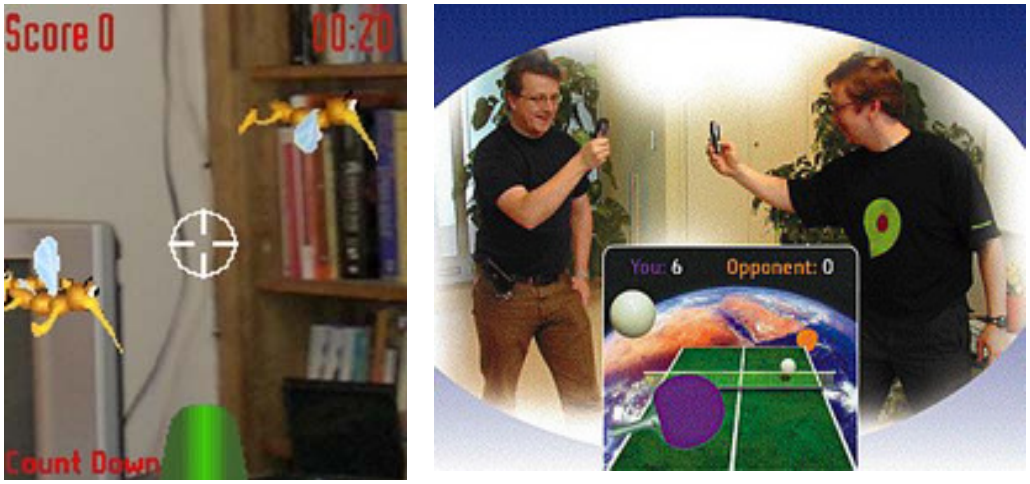


Figure 2-6 Mosquito Hunt and Symball Game

KickReal Foot-based Mobile Interaction with Games

It is also possible to use blob tracking to create interesting mobile phone games. For example, the KickReal [24] game uses blob tracking and background subtraction to allow players to use their real feet to kick a ball into a virtual soccer net. The user plays by

holding the phone parallel to the ground and looking down at the phone screen. The camera on the back of the mobile device is used to detect kicking movements from the user's feet. When the user's foot intersects and a ball shown on the screen the virtual ball is sent towards the goal. KickReal is written in C++ and runs only on Nokia Symbian Series 60 phones.

Figure 2-7 KickReal – showing a real foot and virtual ball

Tiny Motion library

A number of tools are being developed to assist in producing simple image processing applications. The Tiny Motion library [52] is a C++ library for camera phones that use the BREW operating system. TinyMotion detects the 2D movements of mobile phones in real time by analyzing image sequences captured by the built-in camera. The movements are categorized into horizontal, vertical, rotational and tilt motion. This technique does not require additional sensors or special backgrounds to assist with the movement detection. The algorithms in TinyMotion have four major steps, 1.Colour space

conversion. 2. Grid sampling. 3. Motion estimation. 4. Post processing. These processes can be carried out efficiently by integer operations. The TinyMotion library is robust enough to work in a wide range of lighting conditions and backgrounds. Jintao [26] describes the design and implementation of TinyMotion, and also several sample applications that use the library, such as motion controlled Tetris.

Figure 2-8 Using TinyMotion to play Tetris

2.1.3 3D Camera Pose Determination

The most complex image processing stage we are interested in is being able to determine 3D camera pose from a single visual marker in real time. By combining general and 2D image processing techniques, an AR application can be produced. To create these applications it is necessary to track the position and orientation (pose) of the camera providing the video. There are many techniques for doing this, and one of the most

popular uses the ARToolKit library [27] which calculates 3D camera pose from a single square tracking marker.

ARToolKit

In 2003 the ARToolKit library was ported to the PocketPC platform and used in the first fully self-contained PDA AR application [28]. A year later, Möhring created the first AR application [29] that would run on a consumer cell phone. Since mobile phones had scarce processing power at that time, he used a very simple tracking algorithm that would give only a very coarse estimation of the objects position on the screen. Möhring's application showed a simple 3D virtual car model with 500 polygons that was drawn at 4-5 frames per second. Henrysson then ported ARToolKit to the Symbian platform [5] and created a collaborative AR Tennis game [30]. With AR Tennis players see a virtual tennis court superimposed over a piece of paper with tracking squares on it and then hit a virtual ball to each other, using the real phone as a virtual tennis racquet. Bluetooth is used to convey the position of the ball between the two phones and when the phone hits the ball a sound is heard and vibration felt.

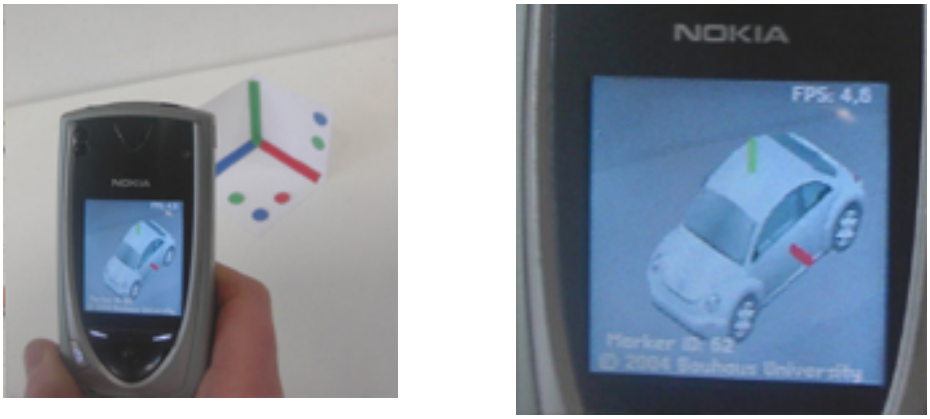


Figure 2-9 Möhring's mobile phone AR application

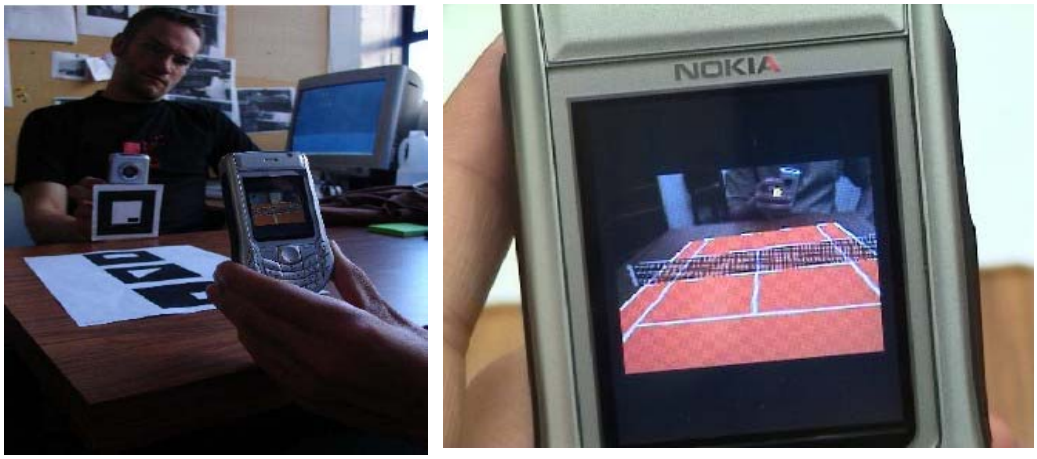


Figure 2-10 AR tennis game

Daniel Wagner's STBtracker

Studierstube Tracker (STBtracker, [31]) is a computer vision library for detection and pose estimation from 2D fiducial markers. Its concept is very similar to that of ARToolKit, but its code base is completely different. STBtracker has been written from scratch with high performance for PCs as well as mobile phones. STBtracker's memory requirements are very low (100KB, 5-10% of the ARToolKit memory footprint) and

processing is very fast (about twice as fast as ARToolKit on mobile phones). This library provides us with a standard to measure other mobile phone AR applications against.

Figure 2-11 Studierstube Tracker

Table2-1 summarizes the computer vision applications presented in this chapter.

Static Image Processing	**2D Image Processing**	**3D Pose Calculation**
QR Code [11] SpotCodes [12] Sema-Code [13] PullFace [14]	Mosquito Hunt [21] Marble Revolution [22] Symball [23] Rohs' tag recognizer [19] KickReal Foot-based mobile Game [24] TinyMotion [26]	AR Tennis [30] Möhring's Mobile Phone AR Application [29] STBtracker [31]

Table 2-1: Mobile phone computer vision applications

2.2 Existing Computer Vision Libraries

In this section, we describe two computer vision libraries that are currently available for mobile phones. These are the Nokia Computer Vision Library and the Phone Vision Library. These libraries offer more information for us to design low level functions.

The Nokia Computer Vision Library (NCV) was created by the Nokia Research Center. NCV was built on the Symbian OS on S60 and provides additional imaging related functionality to developers to be used in third party software. The library provides the usual image operations including geometric transformation, filtering, colour conversion, and image statistics. As well as the current operations, NCV also includes powerful linear algebra operations for more advanced image applications. These new functions include creating matrices, the use of vectors in arbitrary dimensions, and related operations. High-level image and video analysis functionality such as image pyramid and motion estimation are new features of the library.

The Nokia Research Centre website [32] outlines the features of the Nokia Vision Library (NCV) [33] as shown below.

- Standard operations: resize, rotate, etc; Image properties, statistics, edges, corners, features; Colour conversions and manipulations; Combining images with either optimal quality or speed.
- Motion estimation: determine the camera motion (up, down, left, right, shaking).
- Powerful math functions: Complete linear algebra library integrated.
- Camera physical property measurement.

- Standard building blocks for image processing and computer vision.

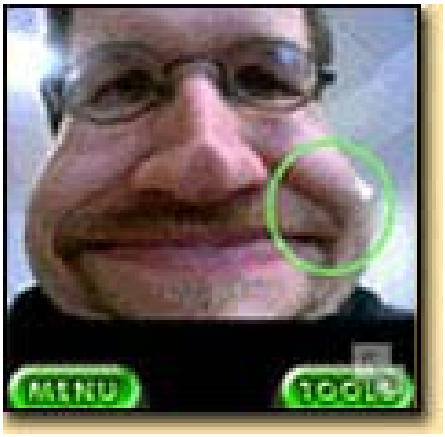


Figure 2-12 Nokia Computer Vision Library application

Phone Vision Library (PVL)

Phone Vision [35] is a basic computer vision library produced for Nokia series 60 devices. The phone vision library was produced by the Bauhaus University. The library incorporates some basic algorithms for edge/corner detection and some higher level algorithms, such as cross-correlation and feature tracking. PVL is automatically loaded as a static interface DLL into the mobile RAM whenever a program that uses it is started. The library is also automatically unloaded when it is no longer required.

One of the applications which use this library is called PhoneGuide [36] which uses static images to perform object recognition in a museum setting. The user is able to walk around a museum taking pictures of exhibits with their mobile phone. The PhoneGuide software performs 2D feature recognition using normalized colour based features to identify the object being photographed and shows additional information about that

object on the phone screen. The photographed museum exhibits can be recognized with a probability of over 90% in less than one second on current phones. The amount of data that is required for differentiating 50 objects from multiple perspectives is less than 6kb.

Figure 2-13 Using the Phone Guide to Get Museum Information

2.3 Existing J2ME Computer Vision Applications

The previous applications were developed using native C or C++ coding and are not portable across phone platforms. However there are examples of J2ME applications for computer vision on mobile phones.

Mobile Landmark Recognition

The Mobile Landmark Recognition application [36] incorporates several of the techniques mentioned above, for use in a mobile device. Using this application the user is able to walk through an unknown environment and take a picture of a specific point of interest, and then be provided with information on this topic. The application was created

in an effort to produce features that were robust, simple and cheap and could be used in an initial selection of candidates for further classification. These features were used in a hierarchical learning scheme and saved to a database through the network. The Mobile Landmark Recognition application was divided in two parts: (1) the mobile application and (2) the database application. The mobile application was developed using the J2ME platform and the database application was built using MATLAB located on a remote server.

Figure 2-14 Mobile Landmark Recognition

Blob Mobile Application

EZ Plus [37] is a small application for mobile phones using J2ME. It uses an efficient method for colour segmentation, based on colour blobs and retrieves information from a database. The interface allows individual blobs to be selected, such as the one highlighted in figure 2-15 with the colour red (the face in the middle). The J2ME platform provides functions for EZ such as controlling the digital camera, rendering 3D points and 2D sprites.

Figure 2-15 Mobile image categorization and retrieval using colour blobs

3 Developing Computer Vision Application for Mobile Phones Using Java

In this chapter, we first introduce the Java 2 Micro Edition (J2ME) (more recently known as Java ME) platform and what tools are needed for developing an application in J2ME platform. Next, we will describe the structure of our computer vision library. Finally we describe how we chose the algorithms included in our computer vision library

3.1 Using Java for Mobile Phone Applications

There are various options for application development for mobile phones, summarized in the chart below. It is fairly safe to narrow down the mobile platforms suitable for computer vision applications to the following four: J2ME, Symbian, .NET (Windows Mobile Smartphone), and BREW.

Platform	Overview
Java ME	Second best reach, best overall development
Flash Lite	Good for graphics-heavy applications in supported markets
Symbian	Strong support from Nokia, best access to hardware
.NET	PocketPC + Windows Mobile Devices
BREW	The only option for CDMA networks
Python	Great for quick prototypes, still immature
WAP	Largest overall reach, lightweight functionality

Figure 3-1 Mobile Development Options [38]

The following charts show relative comparisons between the different development options. J2ME has the largest availability among the four prime candidates by a huge margin and in general has a lower learning curve, extensive developer community, and J2ME games require less time during re-implementation and porting. For these reasons, we focused on what was possible with J2ME.

Platform	Language	X-Platform	Learning Curve	Emulator	Availability
Java ME	Java	Average	Average	Free	~1.5bn
Flash Lite	AS	Excellent	Average	With IDE	77-115m
Symbian	C++	Average	STEEP!	Free	120m
.NET	C#, C++, VB.NET	WM	STEEP!	IDE	4.5m
BREW	C++	CDMA only	STEEP!	Simulator	????
Python	Python	FREE	Gentle	Add-on	Nokia-only
WAP / Mobile Web	XHTML, WML	FREE	Gentle	Free	2bn+

Platform	GUI	Functionality	Phone Data Access	Developer Community
Java ME	2D/3D, Many widgets, Visual Form Builder	Varies by handset, no CellID, high res pics	Varies by handset, Optional APIs	Extensive
Flash Lite	2D/3D, Many widgets, Visual IDE	Partial through API	None	Extensive
Symbian	2D/3D, Many widgets, Visual Form Builder	No restriction	Simulator	Extensive
.NET	2D/3D, Many widgets, Visual Form Builder	Limited audio	Full	MSDN
BREW	2D/3D, Many widgets, uiOne	Operator dependent	Full	Limited
Python	2D Graphics, some widgets	Partial through API	Partial	Small, but growing
WAP / Mobile Web	Basic forms. Inconsistencies	Limited to browser	None	Extensive

Figure 3-2 Generalized Comparisons of the Mobile Development Options [39]

In developing a Java based computer vision application for mobile phones there are two important aspects that must be considered:

- J2ME programming
- Using Java 2D and 3D graphics and multimedia libraries

In this section we review each of these elements first and then describe in detail a sample J2ME computer application that we have developed for a mobile phone.

3.1.1 Developing a J2ME Application

In order to run Java applications on mobile devices it is necessary to code for the J2ME. J2ME is a specification of a subset of the Java platform aimed at providing a certified collection of Java APIs for the development of software for small, resource-constrained devices such as mobile phones, PDAs and set-top boxes.

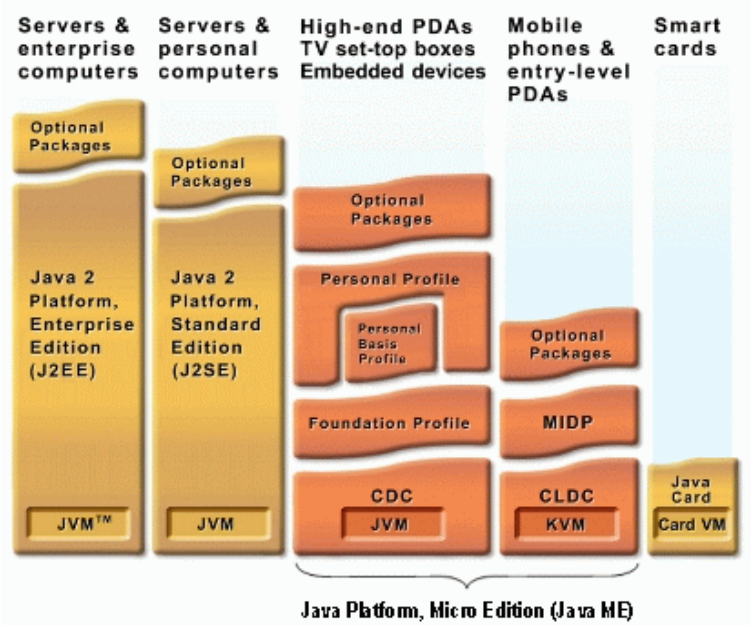


Figure 3-3 The J2ME Platform [8]

The following general information was collected from a variety of sources listed in the Resources section. J2ME is divided into configurations, profiles, and optional API packages. Configurations generally describe the device memory configuration and represent the minimum set of APIs used for developing applications to run on a range of devices. There are currently two J2ME configurations; (1) Connected Device Configuration (CDC) and (2) the Connected Limited Device Configuration (CLDC).

CDC require a minimum requirement of 512kb of ROM (read-only memory) and 256KB of RAM as well as some type of network connection. CDC is typically found on high-end PDAs and other devices more powerful than a mobile phone. CLDC is of more interest for us because it supports mobile phones and other devices of small size (such as pagers). The CLDC configuration requires a minimum of 160kb of ROM and 32kb of RAM, a 16-bit processor, low power consumption, and a network connection (which is assumed will be slow and intermittent).

The reference implementation of CLDC is the K Virtual Machine (KVM) which is based on a small JVM. The KVM does not allow native methods to be added at runtime and only includes a subset of the bytecode verifier. Two versions of CLDC have been deployed, CLDC 1.0 (JSR 30) and CLDC 1.1 (JSR 139) which added floating point data types. A profile sits on top of a configuration and the profile is a more specific set of APIs that further target a device. The main focus is on the Mobile Information Device Profile (MIDP) and there are currently two: MIDP 1.0 (JSR 37) and MIDP 2.0 (JSR 118). MIDP 2.0 offers support for multimedia (java.microedition.media), a game user interface

API (java.microedition.lcdui.game), and many other features that are important for computer vision applications for mobile phones. Fortunately most phones available today and all new phones support MIDP 2.0 (refer to [40] for a J2ME Device Table). MIDP 3.0 (JSR 271) is currently under development.

MIDP hardware minimum requirements are as follows:

- 256KB of ROM for the MIDP API libraries
- 128KB of RAM for the Java runtime system
- 8KB of non-volatile writable memory for persistent application data
- Screen size of 96x54 pixels with 1-bit colour depth (black and white at least)
- Some input device, either a keypad, keyboard, or touch screen
- Two-way network of some type (intermittent is expected)

The optional API packages include functionality that will only be supported on certain devices. Some of APIs more relevant to the types of applications found in Chapter 2 include: Java APIs for Bluetooth (JSR 82), Mobile Media API (JSR 135), Java Binding for the OpenGL ES (JSR 239), and Advanced Graphics and User Interface (JSR 209). There are also OEM specific APIs that provide access to proprietary features and functionality (which can include audio, camera, etc).

MIDP applications are called MIDlets, which can be thought of as Java applets for mobile phones. The diagram below shows the lifecycle of a MIDlet.

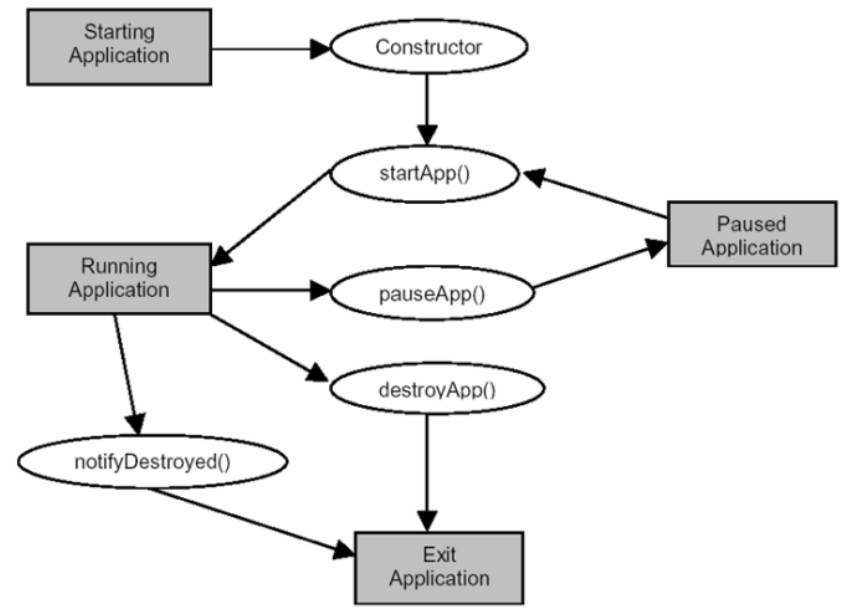


Figure 3-4 MIDlet life cycle [44]

Since J2ME is a scaled down version of J2SE, it has a smaller footprint and doesn't contain heavyweight classes like swing or awt. The J2ME User Interface is based around a succession of screens. The UI elements are not subsets of AWT/Swing. MIDP provides some limited UI elements such as: Form, Alert, Choice and ChoiceGroup, List, StringItem, TextBox, TextField, DateField, Guage, and Ticker. For a good description of how J2ME programming differs from normal Java programming. Brief highlights are:

- No floating point in CLDC 1.0, although CLDC 1.1 includes floating point support. There is a fixed point library for J2ME available for non-commercial use that would be worth investigating further at [43]
- No object finalization – mechanism by which objects can clean themselves up before garbage collection

- No reflection
- No native methods
- No user classloading
- Multithreading similar to J2SE, but no interrupt() method
- The J2ME Math class is a subset of J2SE version, more so with CLDC 1.0 since there is no floating point support
- The Runtime and System classes are greatly reduced from J2SE versions
- CLDC only includes a dozen classes from J2SE's java.util package

In terms of the development work for this report, the IDE used was:

- Java 2 Platform, Standard Edition
- Eclipse SDK 3.1
- EclipseME 1.0
- J2ME Wireless Toolkit (J2ME WTK) 2.2

3.1.2 Using Java 2D and 3D Graphics and Multimedia Library

Most camera applications will need to draw 2D or 3D graphics on the mobile phone screen. There are a number of high level Java libraries that can be used to provide graphics and multimedia (audio, video playback) functionality on the mobile phone. Pulli [41] describes graphics options for mobile phones, including OpenGL ES, a low level API, M3G (also known as JSR-184), and a high level API for Java. Figure 3-5 below shows the relationship between the Java applications, M3G library and the low level

OpenGL ES API. The OpenGL ES and M3G libraries are based on earlier graphics systems designs such as OpenGL, OpenInventor, Iris Performer, VRML, and Java 3D.

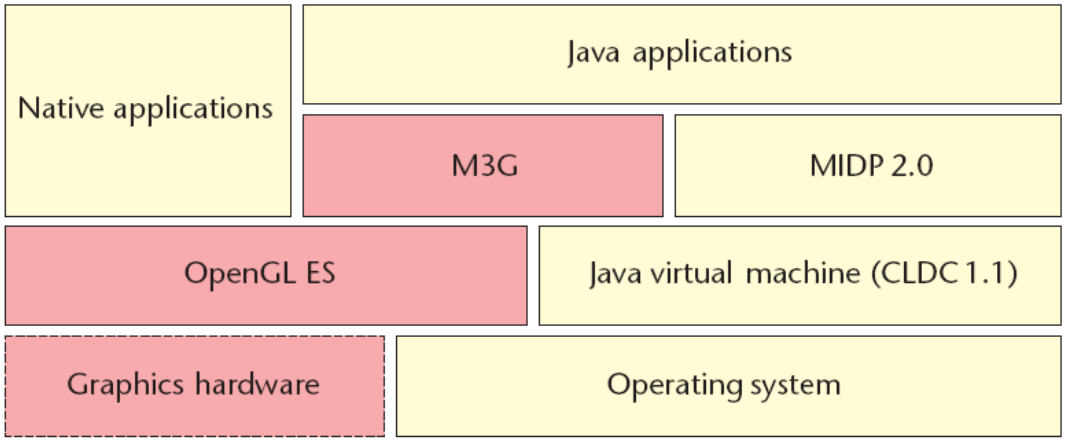


Figure 3-5 Graphic options for mobile phones [41]

The JSR-184 library is designed to provide 3D graphics capabilities to mobile devices that support Java technology. The JSR-184 can be accessed in two different modes: immediate mode and retained mode. In immediate mode, the processes can be compared to the low-level functionality of OpenGL ES and are similar to 3D APIs. If required, the user can define the rendering process by hand. Retained mode hides all low-level functionality from the user, allowing animated 3D content to be loaded and displayed with a few simple lines of code. The two modes affect the internal rendering pipeline of the implementation, and the differences are not apparent to the user. It is possible to load, display, and modify 3D graphics content efficiently in both modes. The format of the 3D data files supported by the JSR-184 is .m3g, which is a very compact but a versatile format, allowing data of nearly any scale to be stored within a single file. We will use both of these models in out project to explore how we can maximize efficiency.

3.2 Structure of Our Computer Vision Library

Although J2ME has been used to develop computer vision applications, no open source computer vision library has yet been developed based on J2ME. We have developed our own computer vision library based on existing non-J2ME computer vision algorithms.

In our library, we focused on developing three types of computer vision applications: general image processing, 2D image processing and 3D camera pose determination. The structure of our library is shown in Figure 3-6. In this thesis we introduce the algorithms which are most relevant to augmented reality application.

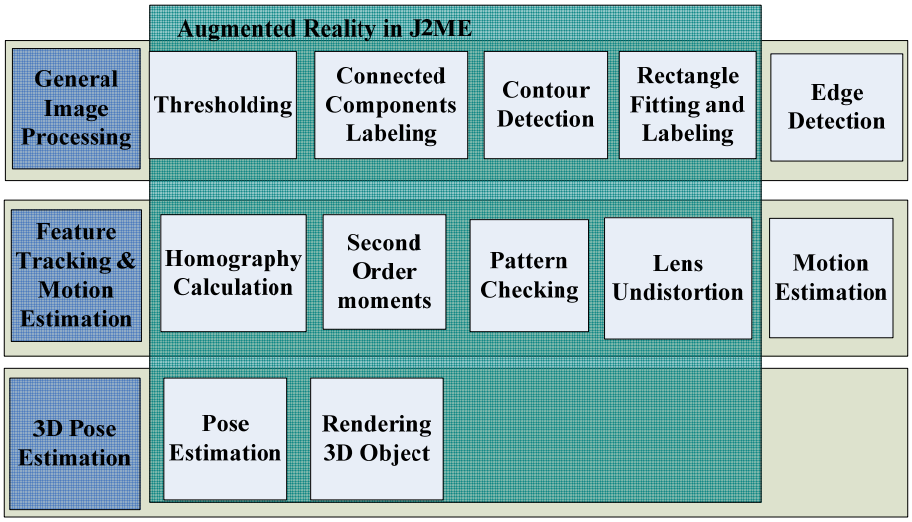


Figure 3-6 structure of our computer vision library

3.3 The Criteria for Choosing the Algorithms

Efficiency:

The most important task for us in the developing computer vision algorithms based on J2ME, is to look for suitable algorithms. The algorithms must be suitable for less powerful processors but must also be accurate. As an example, the QR code has an open source library for J2ME platform. The QR code pattern detection algorithms use three finder patterns to locate a potential pattern candidate and perform more accurately than others but it still takes up to 2-3 seconds to process an image. This is not suitable for a mobile augmented reality application in J2ME.

Portability

Another factor that needs to be considered is the portability of the algorithm. The ARToolKit library is a popular augmented reality application. It has acceptable performance on PC and Symbian operating systems, but ARToolKit is not well documented and hard to understand. There are limited resources describing the image processing algorithms used in ARToolKit, so it is difficult for us to port ARToolKit to the J2ME platform.

There are two existing computer vision libraries: STBtracker and Visual Code. These libraries guided us in building our own computer vision algorithms. We implemented most of the algorithms contained in these libraries, then we compared them in different stages of image processing. Finally we selected the most efficient and suitable algorithms for our J2ME library.

4 Implementation

In this chapter, we focus on introducing the algorithms used in our computer vision library. First we design our marker system. A new marker system is required because the Visual Code marker detection algorithm is error prone (marker detection is very sensitive to lighting changes and the corner points in the marker are hard to detect) and the template matching algorithm used in STBtracker is time consuming (up to 2 seconds).

Our marker consists of the following elements: 2 guide bars, and one cornerstone for determining the location and orientation of the code. There is a black rectangular border outside the guide bar and the corner point. In Figure 4-1, the inner part will be the data area. In our project, we used markers without any data area. This is because it is used for testing only the processing speed of our J2ME application. If the processing speed is satisfactory, we may develop different 3D models for the marker which use different data area in later developments,

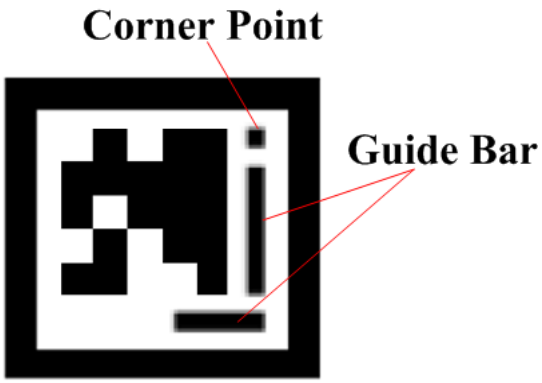


Figure 4-1 The structure our visual marker

In our library, the marker detection module uses an improved algorithm from the Visual Code library. The contour detection and rectangle fitting algorithms from STBtracker library are added to our library to enhance the detection accuracy. We use the homography calculation algorithm from the Visual Code library, because it runs faster than the algorithm from STBtracker. The pose estimation algorithm is from STBtracker. However the original implementations were in C++, and used for complex applications. I have modified the methods to make them suitable for mobile applications, implemented them in J2ME, tested and evaluated the methods using different experiments (see Section 5.1).

4.1 Using the Camera in J2ME

First we discuss the processing of the mobile phone camera inputs using methods in J2ME. In order to use camera functions in J2ME, we need to discuss the Mobile Media API (MMAPI) [42]. “The MMAPI extends the functionality of the J2ME platform by providing audio, video and other time-based multimedia support to resource-constrained

devices. As a simple and lightweight optional package, it allows Java developers to gain access to native multimedia services available on a given device."[42]. MMAPI includes support for a camera, with a special locator "capture://video" used to create its Player. An application can use the VideoControl to display a viewfinder on the screen, and then takes a picture using the getSnapshot function. The default image format is PNG, although the imageType parameter allows other supported formats to be selected. The relevant code is:

```
private VideoControl vc;
player = Manager.createPlayer("capture://video");
player.realize();
vc = (VideoControl)player.getControl("VideoControl");
vc.initDisplayMode(vc.USE_DIRECT_VIDEO, this);
vc.setVisible(true);
vc.setDisplaySize(128, 128);
player.start();
```

Table 4-1: The key part of the code for using camera in J2ME

The above code shows how the camera of a phone is utilized and output displayed on the screen. The camera is executed by using: Manager.createPlayer("capture://video"); this class implements the PlayerListener interface. The realize() method sets the Player into the REALIZED state and makes player ready to receive call backs The initDisplayMode(…) method is used to initialise the video mode that determines how the

video is displayed; this method takes an integer mode value as its first argument with two predefined values USE_GUI_PRIMITIVE or USE_DIRECT_VIDEO. The USE_DIRECT_VIDEO mode is used in this application. With this mode, the video is directly rendered onto the canvas. The region where the video is rendered can be set by the setDisplayLocation() method. By default, the location is (0, 0). The VideoControl can be used to manipulate the size and the location of the video with respect to the Canvas where it will be displayed. Since we are using direct video as the display mode, it is necessary to call setVisible(true) functions in order for the video to be displayed. Finally we start the rendering of the video with the start() method.

4.2 Threshold

In this process, we have two steps to process images. The first step is producing a gray scaled image, and then using the adaptive threshold algorithm for the image.

A Gray Scaled Image

To create a gray scaled image we use similar method described in [19]. The formula *gray = (Red + Green)/2* is used to create a gray scaled version of the colored input image. The blue color component is not included as it produces a poor quality image, especially in the areas of sharpness and contrast. This simple formula is computationally efficient and sets up a sufficient starting point for thresholding the image. Efficiency at this point is vital for the performance of the whole recognition algorithm, as all the image pixels have to be gray scaled.

In J2ME, the default image format is PNG, an where each channel is encoded as an 8-bit hexadecimal number. First, we obtain all the pixels' value from the camera image by using the getRGB( ) method. Then, we will obtain the RGB (Red, Green and Blue) value by using bit-shifting. We modify a true colour image into a grey image by transferring different RGB values to a fixed value. This value is (Red + Green)/2. Table 4-2 shows the key code for producing a gray scaled image.

```
int r = image >> 16 & 0xFF;
int g = image >> 8 & 0xFF;
int m = (r+g) >> 1;
```

Table 4-2: The key part code for producing gray scaled image

After we have obtained a gray scaled image, an adaptive threshold algorithm is taken to produce a black-and-white version image. The adaptive threshold algorithm from visual code [19] was found to be fast and quite suitable for this purpose. We explored the original adaptive threshold algorithm paper [45] and tried to understand and implement this algorithm.

Adaptive Threshold Method

The quality of camera image may not be uniform and the brightness is not regular, so we use the adaptive threshold method to create a black-and-white version of the gray scaled image for the further image processing. The original idea of the adaptive threshold method [45] was to run through the image and calculate a dynamic average threshold

value of the last s pixels. For the next processing pixels, when the value of a pixel is lower than this average it is set to black, otherwise it is set to white. The formulas from [45] are followed:

$$f_s(n) = \sum_{i=0}^{s-1} p_{n-i}$$

where p_n represents the value of a pixel at point n where the image is treated as a single row of pixels, by concatenating the image rows one after one, to form a single array. The function f_s (n) is the sum of the values of the last s pixels at point n. s represents number of pixels which are used to generate threshold value. The value of the result image, T(n) is defined by [45]:

$$T(n) = \begin{cases} 1 \text{ if } p_n < \left(\frac{f_s(n)}{s}\right)\left(\frac{100-t}{100}\right) \\ 0 \text{ otherwise} \end{cases}$$

This value can be either 0 or 1 depending on the percentage difference (t) from the average of f_s (n). Wellner [45] concludes that "using 1/8th of the width of the image for the value of s and 15 for the value of t seems to yield the best results for a variety of images". Wellner [45] also improved the algorithm for calculating f_s. He uses g_s which is a faster way to calculate dynamic threshold value. The idea is to subtract 1/s part of it and add the value of only the latest pixel instead of using all s pixels, as shown in the equation below:

$$\begin{cases} g_s(n) = g_s(n-1) - \frac{g_s(n-1)}{s} + p(n) \\ T(n) = \begin{cases} 1 \text{ if } p_n < \left(\frac{g_s(n)}{s}\right)\left(\frac{100-t}{100}\right) \\ 0 \text{ otherwise} \end{cases} \end{cases}$$

We use above two equations for our J2ME applications. The source code is listed in Appendix A. From my experiment, we use 50 pictures taken by our camera phones to test the most efficient s and t value in our J2ME application. We found using 1/4th of the width and t = 30 will produce better results. Rohs [19] claims that "Gray scaling and adaptive thresholding are considered to be the most time consuming steps in recognition process". In our experiments also, (See section 5.2.2), it turns out that these steps are indeed the most time consuming part in our application, if we do not consider the effect of hardware for taking picture in cell phone.

4.3 Contour Detection

Selecting a contour of a possible area of interest and the black pixels surrounding it, counting them, and then assigning numbers to each one is part of the Contour detection stage [31]. This system of spotting contours is based on the following simple edge finding process which is described in [31]:

"First, scan lines are examined from left to right for edges. White pixels next to black pixels are considered appropriate as square marker border. The algorithm follows this edge until the loop going back to the first pixel is closed or the algorithm reaches the edges of the image. If the loop is closed the contour is saved as a polygon best suited for

the labelling process. To prevent the image border from being reached, the edge is then discarded. Later, to prevent following edges, all pixels that have been visited are marked as processed."

4.4 Rectangle Fitting and Labelling

In the rectangle fitting and labelling stage, the rectangle fitting algorithm described in [31] is put in place. This algorithm is used to determine whether or not the contour candidate has four corner points. Only when there are exactly four corners identified can the contour become a rectangular marker. The example rectangle in Figure 4-2 clearly illustrates this process. The first picture on the left shows the algorithm where an arbitrary point x has been chosen. The farthest distance from x is considered a corner point and labelled C_0. The process calculates the centre of mass from all edge points and a line through C_0 and the new point created is established. Points that are further to the left and right of this line are labelled corner points C_1 and C_2. The algorithm recursively determines additional corner points by constructing more lines using the points C_0, C_1 and C_2. In the example given below, one additional point is found and labelled as C_3. New lines created from C_2 to C_3 and C_3 to C_1 shows that there are no more corners.

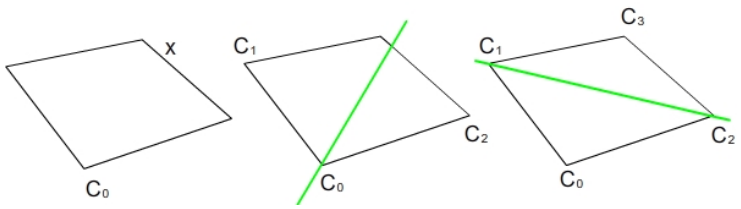


Figure 4-2 Example for fitting a rectangle to a polygon.

A two-step method is commonly used in the labelling process [19]: (1) the interior of the contour discovered earlier is navigated row by row, and (2) each region found is given an initial label. The process may later show that two regions having different labels are actually the same region. If this happens the equal number of the labels is temporarily stored in a table. The multiple labels are then merged and the final label assigned.

4.5 Second order moments

After identifying the shape of the marker, we need to identify it further and determine the orientation of the marker. To identify candidates and orientation bars among the regions found, the concept of second-order moments is used [46].

"The geometric moments of different orders represent different spatial characteristics of the image intensity distribution. A set of moments can form a global shape descriptor of an image." Mukundan [47] described: "The second-order moments are a measure of the variance of the image intensity distribution about the origin. It can be considered as the moments of inertia of the image with a set of axes parallel to the image coordinate axes. The major and minor axis of each region is determined from these moments. The ratio of the lengths of these axes is a good measure of the 'eccentricity' of a region". The rotation of the marker can also be identified using second order moments. The four corner points of the contour will be mapped when the rotation value is known. The detail of the algorithm is as follows:

For each black region, we calculate the centre of image region (x_0, y_0). Then we use formula [19] to calculate the central moments.

$$\mu_{xx} = \frac{1}{|R|} \sum_{(xy)\in R} (x - x_0)^2$$

$$\mu_{yy} = \frac{1}{|R|} \sum_{(xy)\in R} (y - y_0)^2$$

$$\mu_{xy} = \frac{1}{|R|} \sum_{(xy)\in R} (x - x_0)(y - y_0)$$

The central moments μ_{xx}, μ_{yy} give the variances about the centroid. The convariance measure is given by μ_{xy}. Finally we can calculate the angle of silhouette image by using :

$$\theta = \frac{1}{2}\tan^{-1}\left(\frac{2\mu_{xy}}{\mu_{xx} - \mu_{yy}}\right)$$

The above formula is from [47].

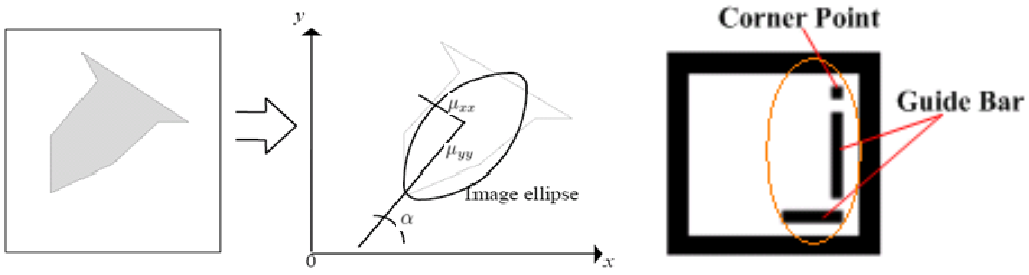


Figure 4-3 An image and the corresponding image ellipse.

From the algorithm of second moments, we can identify our marker and calculate the orientation of our marker (Figure 4-3). The algorithm is much more efficient and simpler than the template matching algorithm. The source code is listed in Appendix B.

4.6 Homogeneous Transformation

A homogeneous transformation is the projection from one plane through a point onto another plane [48]. In this step we use a projective mapping algorithm to calculate the homogeneous transformation matrix. The main principle behind projective mapping is to take points from image plane and project them into the real world. (Figure 4-4 shows the homogeneous transformation) “Homogeneous transformations are used commonly for 3D affine modelling transformation and they are also used for perspective camera transformations.” [48]

The homogeneous transformation can be calculated when four matching points are known. The general form for a projective mapping is given by [79]:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

$$x = \frac{ax' + by' + c}{gx' + hy' + i} \quad y = \frac{dx' + ey' + f}{gx' + hy' + i}$$

In this step we use a 2D projective transformation matrix. In the next step, pose estimation, we use this matrix to calculate the 3D transformation matrix. The first equation includes nine entries (a,b,c,d,e,f,g,h,i), but is defined only up to scale. Thus the total number of degrees of freedom in a 2D projective transformation is 8. So [48] defines i=1 except in the special case that source point (0,0) maps to a point at infinity.

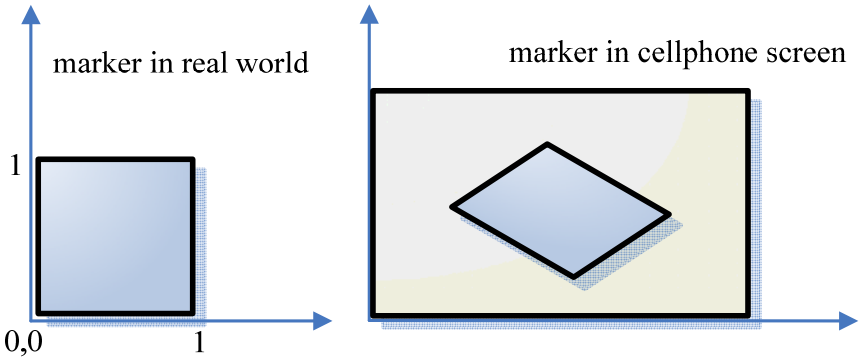


Figure 4-4 .Homogeneous transformation

x' y' is a unit square, so the correspondences points are as follows:

x	y	x'	y'
x_0	y_0	0	0
x_1	y_1	1	0
x_2	y_2	1	1
x_3	y_3	0	1

The parameters a to h are calculated from the four reference points (x_1,y_1), (x_2,y_2), (x_3,y_3), and (x_4,y_4). The following formula from [79] is followed:

$$c = x_0$$

$$a + c - gx_1 = x_1$$

$$a + b + c - gx_2 - hx_2 = x_2$$

$$b + c - hx_3 = x_3$$

$$f = y_0$$

$$d + f - gy_1 = y_1$$

$$d + e + f - gy_2 + hy_2 = y_2$$

$$e + f - hy_3 = y_3$$

[79] defines:

$$\Delta x_1 = x_1 - x_2$$

$$\Delta x_2 = x_3 - x_2$$

$$\sum x = x_0 - x_1 + x_2 - x_3$$

$$\Delta y_1 = y_1 - y_2$$

$$\Delta y_1 = y_3 - y_2$$

$$\sum x = y_0 - y_1 + y_2 - y_3$$

So the final equation is:

$$g = \frac{\sum x \ \Delta y_2 - \sum y \, \Delta x_2}{\Delta x_1 \, \Delta y_2 - \Delta y_1 \, \Delta x_2}$$

$$h = \frac{\sum y \ \Delta x_1 - \sum x \, \Delta y_1}{\Delta x_1 \, \Delta y_2 - \Delta y_1 \, \Delta x_2}$$

$$a = x_1 - x_0 + gx_1$$

$$b = x_3 - x_0 + hx_3$$

$$c = x_0$$

$$d = y_1 - y_0 + gy_1$$

$$e = y_3 - y_0 + hy_3$$

$$h = x_0$$

$$i = 1$$

We use above equations to calculate the homogeneous transformation matrices, which are essential for the pose estimation calculation.

4.7 Pose Estimation Algorithm

This is the part of our library when the estimation of the camera pose is determined from the corner points. Since the pose estimation algorithm involve lot of complex algorithms and can be the most time consuming in terms of development, we use the pose estimation algorithm from STBtracker [31], as we mentioned in the early part of the section 4.

The original algorithm use Gauss-Newton iteration in the pose estimation step and it calculates a preliminary pose from the homogeneous transformation matrices calculated earlier. The algorithm attempts to refine the 6 parameters (3 for position and 3 for rotation) that constitute the camera's pose. The algorithm calculates the Jacobian matrix for the current parameter values at each step of the iteration.

We translate the original C++ STBtracker code to J2ME. Wagner [31] explains "Although this initial pose could be used for tracking, it is coarse and jitters heavily." In order to increase the processing speed and make our J2ME application run in real time we decide to remove the algorithm refining part. We only calculate the initial pose from homogeneous transformation matrices according the method shown in [31]. First, we create a scale matrix to normalize the homogeneous matrices. Then we scale back the homogeneous matrices to the marker size of 60*60 mm. We then multiply inverse camera matrices with this scaled homography matrix. Finally we calculate the normalizing value from the translation and the 3rd column value and obtain the final pose estimation matrices. Appendix C shows the source code of calculating the pose estimation matrices.

4.8 Rendering 3D Objects

The final stage is to combine a 3D object in our frame. We have created two 3D objects for our application. One is a colour cube and the other is a ball with texture on it. The JSR 184 standard has its own format, called M3G (described in Section 3.2). “This is a very versatile 3D format, which can hold a large amount of data such as models, lights, cameras, textures and even animation.” [73]. All rendering processes in JSR 184 are performed with the Graphics3D object. It can even store camera and light information, if the image is being rendered in the immediate mode.

To render with a Graphics3D object, we must first bind it to a graphics context. A graphics context is a particular graphics object that is drawn into something else. By working on the main graphics object we can render directly to the screen, which is what we want to do here. We can obtain a Graphics3D object by calling the Graphics3D.getInstance() method.

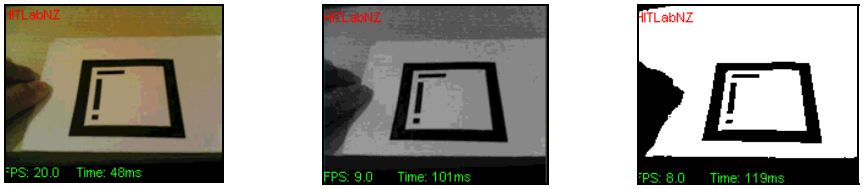
When our world node is ready for rendering, all we need is a camera to view our world --- a rotating cube. A camera in JSR 184 is defined by the Camera class; this class makes it very easy to manipulate the camera in our 3D application with simple translation and orientation methods. The two methods used in this application are translate(float, float, float) and setPerspective(float, float, float, float). The first method simply moves the camera in 3D space by an offset in x, y and z. The second method constructs a perspective projection matrix, and sets that as the current projection matrix. After initializing a 3D world we use the MeshCreator class to create an object, and place a

texture on it. In another application, we use 64*64 low resolution images to the texture the ball.

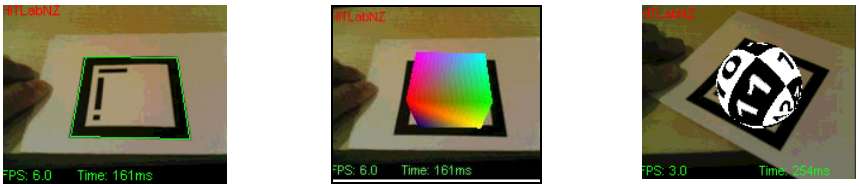
```
g3d = Graphics3D.getInstance();

world = new World();

temptime = Stime;

camera = new Camera();

world.addChild(camera);

float w = getWidth();

float h = getHeight();

camera.setPerspective(60.0f, w / h, 0.1f, 1000f);

camera.translate(0.0f, 0.0f, 5.0f);

MeshCreator mc = new MeshCreator(tex);

m = mc.createTexturedMesh();

world.addChild(m);

world.setActiveCamera(camera);
```

Table 4-3: The key part code for rendering 3D object

Figure 4-5 shows the steps involved in creating a 3D mode in cell phone.



(a) Original image; (b) Gray scale; (c) Threshold



(d) Rectangle fitting; (d) Final application; (e) virtual ball

Figure 4-5 The steps involved in creating a 3D image

4.9 Edge Detection

The process of finding sharp contrasts in intensities in an image is called edge detection. This considerably reduces the amount of data in the image while still retaining its most critical structural features. In conventional images, edges identify object boundaries and are therefore useful for segmentation, registration, and identification of objects.

Sobel Edge Detection Algorithm

The Sobel edge detection algorithm [49], [50], is the most accepted and widely used edge detection algorithm. Compared with other detection algorithms, the Sobel algorithm shows superior performance. It is the one of the most commonly used current image processing technique..

The Sobel detection operator is a common first order edge detection operator that finds contrast by a method comparable to differentiation. The Sobel operator performs a 2D spatial gradient measurement on an image. Primarily the operator is used to discover the approximate absolute gradient magnitude at each point in an input grayscale image. A pair of 3x3 convolution masks are used, one estimating the gradient in the x-direction (columns) and the other estimating the gradient in the y-direction (rows). Figure 4-6 illustrates the Sobel masks.

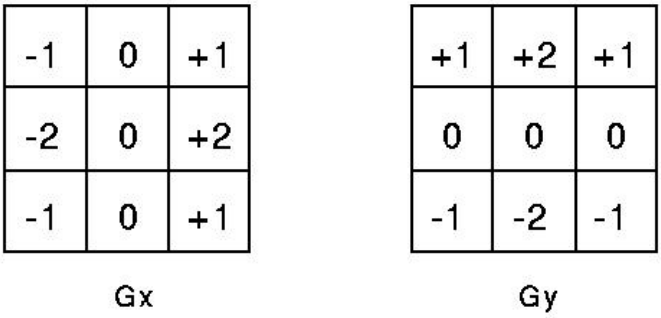


Figure 4-6 Sobel masks [50]

The edges in the horizontal direction (rows) are indicated on the Gy mask and the edges in the vertical direction (columns) are indicated on the Gx mask. The resulting output detects edges in both directions after taking into account the scale of both. The magnitude of the gradient is then calculated using the formula from [50]:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

An approximate magnitude can be calculated using:

$$|G| = |Gx| + |Gy|$$

The Sobel masks [50] describe the algorithms: the mask is slid over an area of the input image altering that pixel's value and then moving one pixel to the right. This process continues to the right until it reaches the end of the row where the process continues at the beginning of the next row. Figure 4-7 shows the mask over the top left portion of the input image as indicated by the green outline. The formula then shows how a specific pixel in the output image is calculated. The centre of the mask is positioned over the pixel being manipulated in the image. Pixels in the last rows and columns cannot be manipulated by a 3x3 mask because if the center of the mask is placed over a pixel in the last row or column, the mask will already be outside the image boundaries.

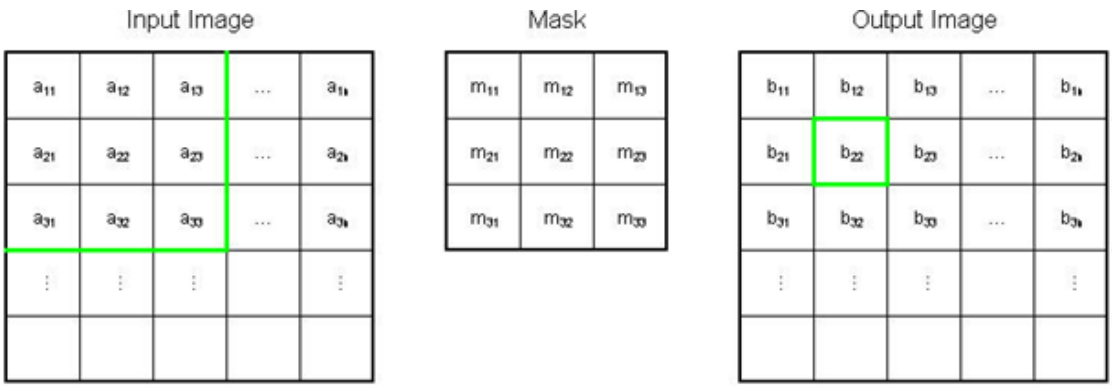


Figure 4-7 The mask being slid over the top left portion of the input image represented by the green outline [50]

The gradient value of each pixel is gained after searching all the pixels of the image. The threshold is set to a value to *(Red+Green)/2*. The gradient value is then compared to the threshold value and an edge is detected when the threshold is exceeded. The edge detection application, which runs on the J2ME simulator, is shown in Figure 4-8.

Figure 4-8 Sobel Edge detection

Canny Edge Detection Algorithm

In order to determine which was more suited to the J2ME library, the Canny edge detection algorithm [51] was also implemented along with the Sobel edge detection algorithm. Three features emphasise the efficiency of the Canny edge detection algorithm: first, is its low error rate, because it is important not to miss edges occurring in images and that there should be no responses to non-edges; second is localisation of edge points, meaning that the distance between the edge pixels found by the detector and the actual edge is at a minimum; and third, is having only one response to a single edge. The third criterion was implemented because the first two do not completely eliminate the possibility of multiple responses to an edge.

The details of how the Canny Edge Detection Algorithm works is described in [76]:

1. use Gaussian smoothing.
2. use the Sobel operator.
3. use non-maximal suppression.
4. threshold with hysteresis to connect edge points.

First, before locating and detecting any edges, filter out any noise in the original image. The Gaussian filter can be computed by using a simple mask used exclusively in the Canny algorithm. Once an appropriate mask has been calculated, the Gaussian smoothing algorithm can be used with regular convolution methods. This convolution mask is normally smaller than the actual image and the mask can therefore easily slide over the image, manipulating one square of pixels each time.

Second, use the Sobel operator described in section 4.9 to obtain the value for the first derivative in the rows (Gy) and the columns (Gx). When edge direction is recognized, relate this edge direction to a direction that can be traced in an image.

Non-maximum suppression, a function used to trace along the edge in the edge direction and suppress any pixel value (sets it = 0) that is not recognized as an edge, may be applied when the edge directions are identified. The output image in this process is a thin line.

Nixon [76] described the last step of Canny Edge Detection Algorithm is hysteresis. It is used to track along the remaining pixels that have not been suppressed. Two thresholds are used in hysteresis. If the magnitude is below the first threshold, the pixel value is set to zero (a non-edge). If the magnitude is above the high threshold, it is made an edge. And if the magnitude falls between the two thresholds, the pixel value is set to zero unless there is a path from this pixel to a pixel with a gradient above the second threshold.

Figure 4-9 shows the Canny edge detection application. The J2ME implementation takes approximately one second to process an image, double the time required by the Sobel algorithm on the same image.

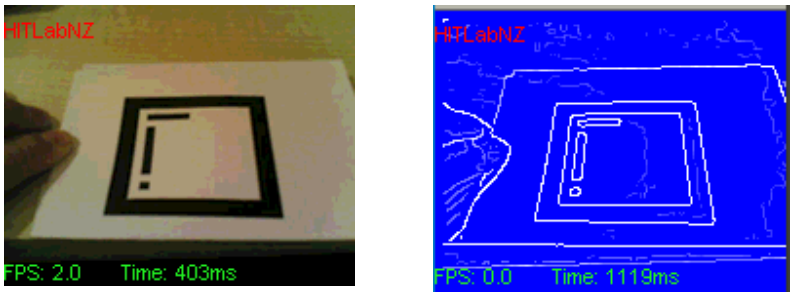


Figure 4-9 Canny edge detection

4.10 Motion Estimation

Motion estimation is the process of identifying Motion Vectors (MV) so that a frame sequence can be depicted in a more compact form as a reference image as well as a displacement vector over time. It allows for the conversion from one 2D image frame to another. Most video compression standards involved with the efficiency and performance of motion estimation, use a modified version of the block matching algorithm (BMA) [54] as tool of choice. The block matching algorithm is based on the assumption that all pixels in a single block experience the same 2D translational motion. The current frame is divided into pixel blocks with motion estimation being performed independently for each pixel block.

Motion estimation is achieved by pinpointing a pixel block from the next frame that best matches the block whose motion is being estimated. The reference pixel block is generated by dislodgment from the current blocks in the next frame. Motion Vector (MV), a pair of coordinates (x,y) showing horizontal and vertical displacement values, provides for this dislodgement

Full Search (FS) Algorithm

The matching process uses most of the computational effort for BMA. The Full Search (FS) algorithm is used to search the whole search region for a block. “If more than one block produces a minimum block distortion measure (BDM), the FS algorithm then selects the block which contains the MV that has the smallest magnitude. The BDM of image blocks may be measured using various criteria such as, the mean absolute error (MAE), the mean square error (MSE), and the matching pel-count (MPC).” [77]

In the centre of the search region Checking Points are used in a spiral track. If the greatest displacement of a MV in both the horizontal and vertical directions is ± d pixels, then the total number of search points used to find the MV for each block can be as high as $(2d + 1)^2$. The FS algorithm provides the most accurate and precise outcome but is very time-consuming. Other faster search algorithms have been put forward to lessen the complexity of FS but these new techniques have been discounted due to lack of search accuracy. The most favoured of these new algorithms is the simple but effective three step search (TSS).

The Three Step Search (TSS) Algorithm

TSS [53] divides the search operation into three steps and refines the search results of the following step in accordance with the previous search. In the J2ME computer vision library, the TSS algorithm is the tool used to start the motion estimation application.

The website [54] describes the TSS algorithm: The first step evaluates nine candidates with a step size of 8 pixels as shown in 4-10. The second step is reduced by half and the search centre is altered to be the best matching candidate from the first iteration. The process continues until the step size becomes equal to one pixel, which is the last iteration of the three-step algorithm. The final candidate selected is the best matching candidate from this iteration and the MV matching this candidate is chosen for the current block,

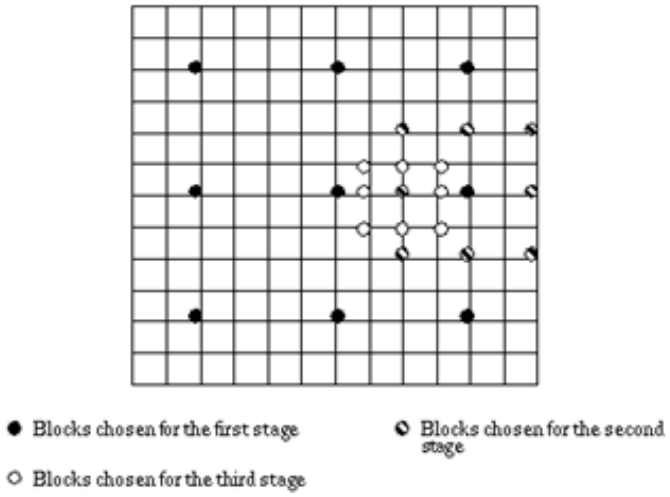


Figure 4-10 Three step search [54]

In our algorithm we search every eight pixels, and then search every four pixels. Finally we search neighbouring pixels using the following code:

search_core(data_block, buf_ref, 8, indexbuf_ref);

search_core(data_block, buf_ref, 4, indexbuf_ref);

search_core(data_block, buf_ref, 2, indexbuf_ref);

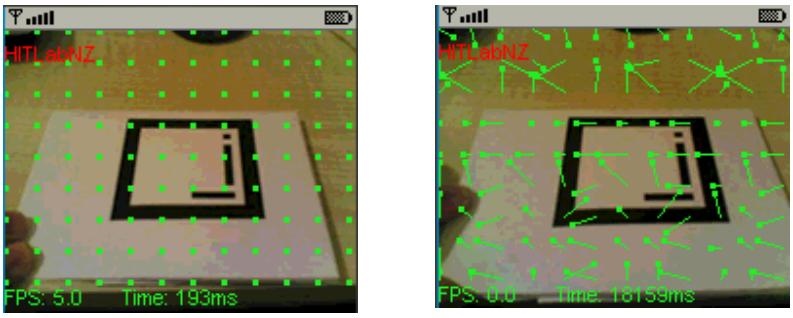


Figure 4-11 Full Search Algorithm

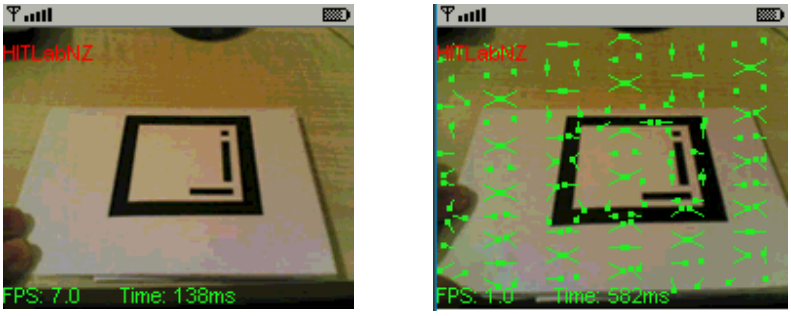


Figure 4-12 Three Step Search Algorithm

Figure 4-11 shows our motion estimation application based on the FS algorithm. Figure 4-12 shows our motion estimation application based on the TSS algorithm. The green arrows show the movements of the pixels. The resolution used for this application is 176*208 pixels. The FS application takes 18s to process the motion estimation algorithm and draw the green arrows on the screen of the simulator, while the TSS only takes 0.5s. However, the FS application shows more accurate result compared to the TSS application.

5 Applications and Experiments

In this chapter, we describe several applications and experiments testing our algorithms.

5.1 Test the Portability of the J2ME Application.

The first experiment we carried out was to test the portability of J2ME application. We used both existing J2ME applications and our new applications to conduct this experiment. The big advantage of using J2ME compared to native code on mobile phones is that it should be portable across different phone handsets and operating systems.

5.1.1 Simple Video Camera Program Testing

To see how many cameras could be accessed from different phone models without code changes, we first used a simple camera program to evaluate the portability of J2ME that utilizes byte-code. The program has a class called “cameraMidlet”. The cameraMidlet creates the cameraCanvas class which requests a video capture player from the MMAPI Manager class, and receives an object implementing the player interface. From this object it requests a video control, and receives an object implementing the VideoControl interface. The developer doesn’t know (or need to know) the actual class of either of these objects, as we will interact with them only through these interfaces. Figure 5-1 shows the simple video camera class diagram.

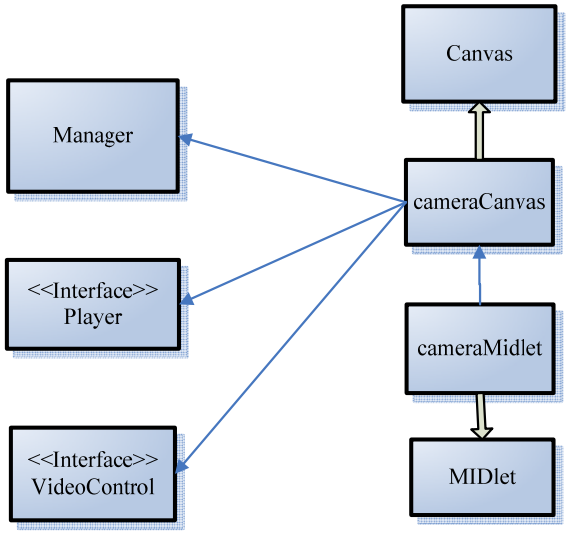


Figure 5-1 Simple video camera program class diagram

In order to test portability we asked for volunteers who could let us use their mobile phones to run the application on. Of the sample pool available to us, most devices were from Nokia or Sony Ericsson. Most phones ran this application successfully, but some of them made a confirmation 'click' sound after a picture was taken. This sound could be suppressed in some phones such as Nokia N72 and 6680, but not for other phones such as N95. Most phone spent 300-400ms to take a picture. But the N95 spent around 6 seconds to take a picture and show it on the screen. Samsung D508 and one windows mobile phone could not run this application.

Mode:	Operating System	Simple Video Camera Application	Graphics Testing (FPS)	Our Application	
				Original picture(ms)	Processed images(ms)
Nokia 6680	Symbian OS 8.0A	success	10fps - 32fps	890-1070	1200-1400
Nokia 6630	Symbian OS 8.0	success	9fps - 30fps	1000-1200	1240-1400
Nokia 5700	Symbian OS 9.2	success	20fps - 35fps	Not tested	Not tested
Nokia N95	Symbian OS v9.2	success	25fps - 35fps	7600-8200	8000-9000
Nokia N91	Symbian OS 9.0	success	20fps - 30fps	7400-8230	8100-9000
Nokia N70	Symbian OS 8.1	success	10fps - 30fps	Not tested	Not tested
Nokia N73	Symbian OS 9.1	success	17fps - 30fps	Not tested	Not tested
Nokia E65	Symbian OS 9.1	success	15fps - 30fps	7600-8200	7600-8200
Motorola E680i	Linux	success	17fps - 30fps	790-1170	1200-1400
SonyEricsson K610	NO	success	failed	Not Tested	Not Tested
SonyEricsson V630	NO	success	failed	Not Tested	Not Tested
SonyEricsson W810i	NO	success	10fps - 30fps	940-1090	1230-1400
SonyEricsson W800	NO	success	10fps - 30fps	940-1090	1200-1530
Samsung D508	NO	failed	10fps - 20fps	failed	failed
Imate Jam Windows mobile	Windows Mobile	failed	Failed?	failed	failed

Table 5-1: Results from the test applications described in Sections 5.1.1, Sections 5.1.2 and Section 5.1.3

5.1.2 3D Java Graphics Capability Testing

The second experiment evaluated the graphics capability of the mobile phones. A program called 3D Maze [55] was used as the test program. This 3D game is built using the M3G library. Most of the sampled mobile phones supported this game. In some phones, the FPS value would drop to around 10 if the scene was composed of complex polygons, where in a simpler scene the FPS value would be around 32. Most of mobile phones tested could run this type of complex 3D game fluently. (Refer to the table 5-1.)

In the game, the player sees the world from a traditional, first-person point of view and walks around a maze. The objective was to solve the maze in a minimum amount of time. The objects in the game are basically the floor and the maze walls. Two labels were also added to indicate the start and end of the maze. The end marker is animated, moving up and down the y axis. The scene has a background sky simulating a star field. The player is located where the main camera is, as she or he moves around the maze. The images in Figure 5-2 also show a "FPS" number at the bottom of the screen. This reports the number of frames per second, which is used to test the graphics speed of the J2ME library to decide whether it is suitable for further development.

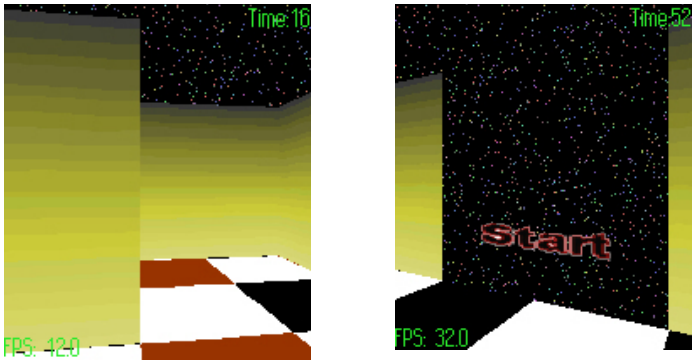


Figure 5-2 3D Maze Game

5.1.3 Our Application 1 (Combining Camera and 3D object)

We developed our first test application for combining camera input, image processing, and 3D object viewing. This application helps to further evaluate the computer vision capability of mobile phones using J2ME. In this application, we have made a 3D cube textured with real-time images to compare the processing speed between the original picture and the picture after image processing. We have provided two options: keypad input “5” creates the cube textured with processed images, and keypad input “6” applies texture based on original images. The program flow is displayed in Figure 5-3.

The time required to process an original picture textured on the cube was around 1000 ms, while adding the image processing steps of changing to grayscale and making transparent images increased this only slightly, to around 1200ms. The method for producing gray scaled image is described in section 4.2. The method for producing transparent image is similar to the method of producing gray scaled image. The detail of this method is described in [44]. A spot light effect was added to test the 3D graphic processing time.

Note: in this application, the image processing time is around 0.22 seconds. This suggests that the most expensive step for current applications is the time taken to access each image from the camera .Figure 5-4 shows our application running on Nokia 6680. The results are summarized in Table 5-1.

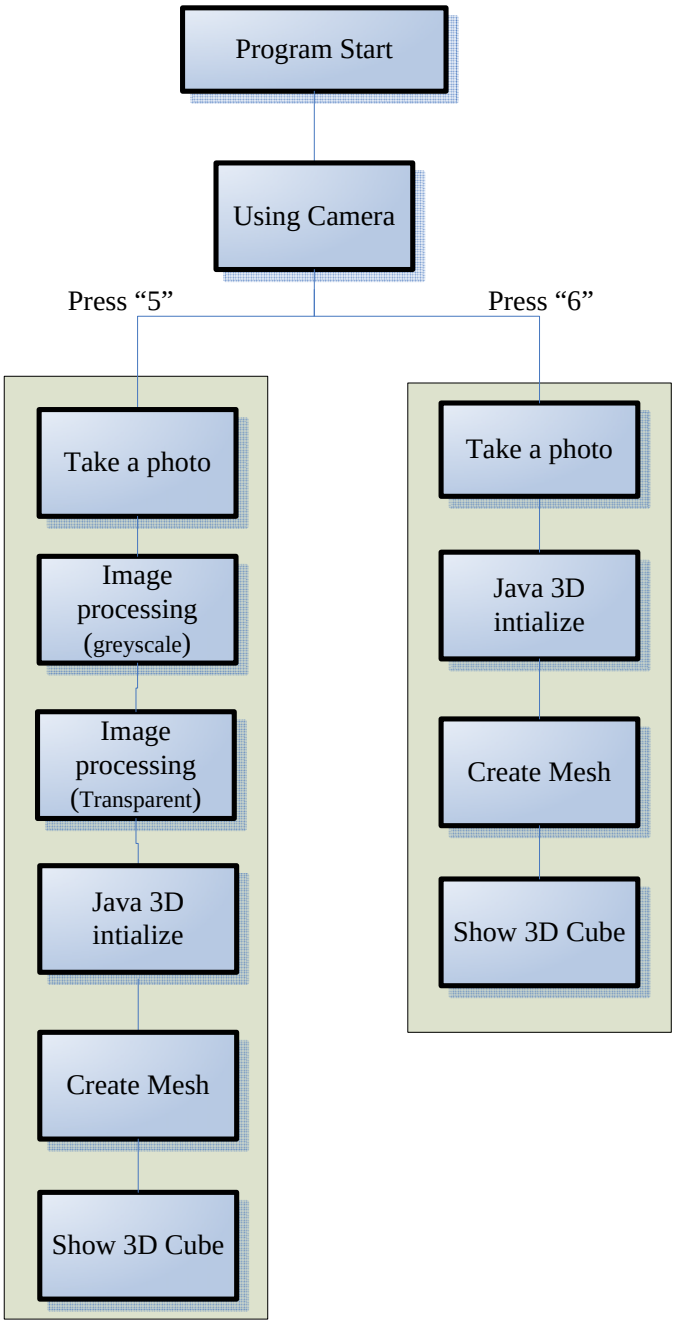


Figure 5-3 The program flow

a) Cube with texture of original images

b) Cube with texture of processed images

Figure 5-4 The test application running on a Nokia 6680.

5.2 Testing Our AR Application

In this section, we use different experiments to test the efficiency of our augmented reality algorithm.

5.2.1 Testing Our AR Application on Different Phones

We implemented a J2ME augmented reality application by using the pose calculation algorithm from our computer vision library. We have made a 3D cube and a 3D ball with a texture on it. The cube consists of fewer polygons than the ball and thus takes less rendering time. However, the more complex the 3D scene, the more time is required to draw the virtual content. We used a simple 3D object to test the feasibility of our application in J2ME.

Evaluating the performance of the application, most of the time seemed to be spent on capturing the video image; the overall average processing time was between 0.55 ~ 1 second. The Nokia N95 was the most modern mobile phone tested, but took the most time to run our application (nearly 7 seconds to process the algorithms).

In order to increase the processing speed, we tried to decrease the image resolution. We reduced the resolution to 100*80 pixels, and set this parameter when using camera to take a picture. We use VideoControl's getSnapshot() method in the J2ME library to achieve this feature. The getSnapshot() method returns an array of bytes, which is the image data in PNG format. Then we create an Image from them so we can show the user the picture just taken. The following code is used:

```
byte []raw;
try{
        raw = vc.getSnapshot("width=100&height=80");
        mTextureImage = Image.createImage(raw, 0, raw.length);
}
catch(Exception e){
        System.out.println(e.getMessage());
        System.out.println(e);
}
```

However, the results were not what we expected. There was no substantial decrease in processing time, and some devices such as the Nokia N72 had some unhandled errors (Refer to table 5-2). We explore developer forums to try to find out the reason for these unhandled errors. We were able to find no suitable answers for this problem, although a possible reason would be that the programming API for different phones is different. For example, In the above code if we change the code to “raw = vc.getSnapshot(null);”, the unhandled error disappeared.

Therefore image resolution is not the key to determining the efficiency of the application. Figure 5-5 shows our application running on two mobile phones and Table 5-2 shows the result of the test for our application on different phones.

Figure 5-5 The experimental AR application running on the Nokia N95 (left) and Nokia 6680 (right)

Mode:	Operating System	Original Resolution (X*X)		Resolutions of 100*80
		FPS	Time (ms)	Time (ms)
Nokia 6680	Symbian OS 8.0A	1 – 0	550-650	Failed
Nokia 6630	Symbian OS 8.0	1 - 0	900 – 1100	Failed
Nokia N72	Symbian OS 8.1	1 – 0	520-650	Failed
Nokia N95	Symbian OS 9.2	0	6000 -8000	6000-8000
SonyEricsson W800	NO	0	2000-2700	2100-2600
SonyEricsson W530	NO	Failed	Failed	Failed

Table 5-2: Performance of the experimental AR application

5.2.2 Testing the Efficiency of Each AR Algorithm

The most time consuming stage of our applications was when J2ME calls the camera function to take a picture and display it on the screen. In order to analyse the processing progress, we separated out each algorithm to test the efficiency. We tested each algorithm across several phones to calculate the average time spent for each algorithm. Table 5-3 shows the results of different phones.

The N95 took 6 seconds to take a photo and spent 90% of the total time in our application. Others phones spent 30% of the total time to take the picture. Thresholding is another step which takes significant time; around 27% of the total processing time. M3G was the fastest method of making a virtual 3D object, because the object is initialized in the beginning and for each frame, and the object only needs to change position and scale. Figure 5-6 shows the time spent for each algorithm and Figure 5-7 shows the time percentage for each algorithm

Algorithms:	**Nokia 6680**		**Nokia 6630**		**Nokia N72**		**Nokia N95**		**SonyEricsson W800**	
	Time (ms)	**%**	**Time (ms)**	**%**	**Time (ms)**	**%**	**Time (ms)**	**%**	**Time (ms)**	**%**
Take picture in camera	233	38%	366	40%	233	39%	6530	90%	500	23%
Threshold	165	27%	199	22%	165	27%	250	4%	375	17%
Contour Detection	55	9%	77	8%	52	9%	47	1%	210	10%
Rectangle Fitting	55	9%	77	8%	55	9%	60	1%	230	10%
Second Order Moments	30	5%	60	4%	26	4%	30	1%	210	4%
Homography Calculation	20	3%	56	6%	20	3%	25	1.%	200	9%
Pose Estimation Algorithm	32	5%	50	5%	32	5%	27	1%	240	11%
Rendering 3D Objects	20	3%	40	4%	20	3%	30	1%	230	10%

Table 5-3: Summary of performance of each of algorithms which runs in different phones

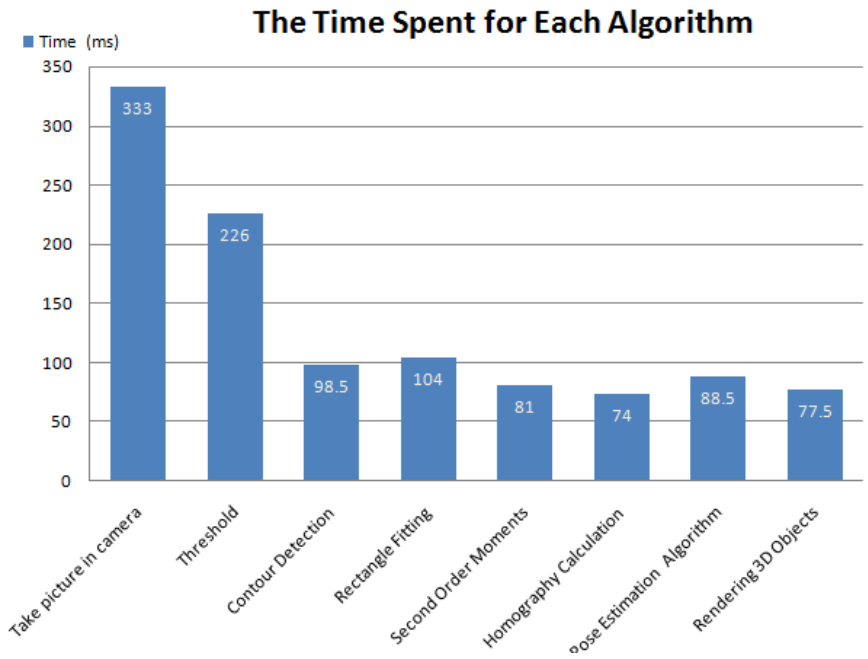


Figure 5-6 The time spent for each algorithm

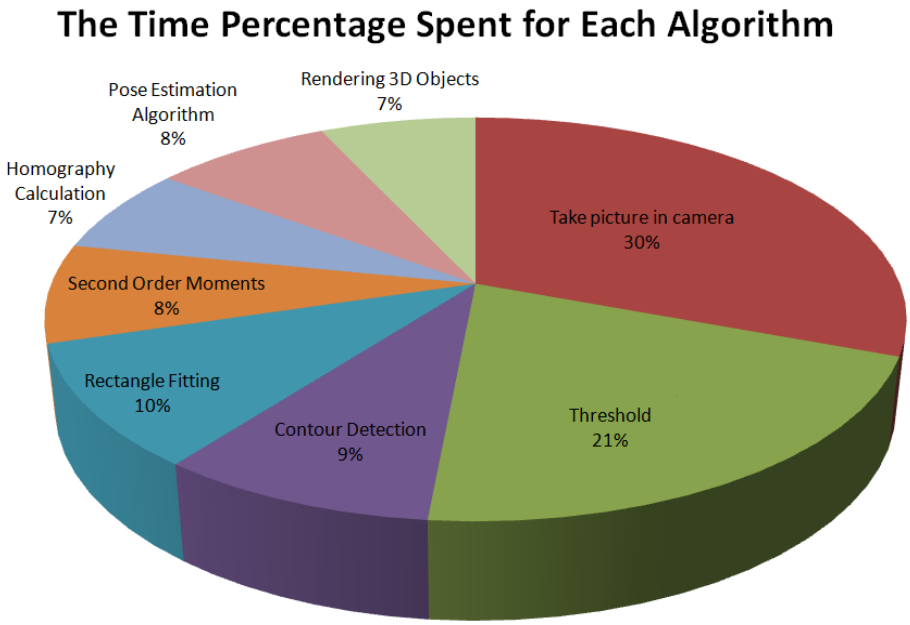
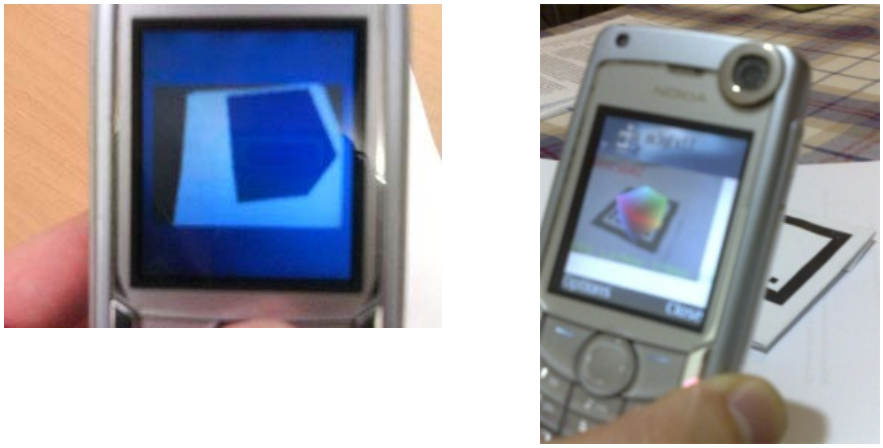


Figure 5-7 the time percentage spent for each algorithm

5.2.3 Comparing the J2ME AR application with a Symbian C++ AR application

These results can be used to compare the performance of our experimental AR application (Section 5.2.1) with the ARToolKit native C++ AR application. The C++ application was developed by Henrysson [56], and rendered a simple 3D blue cube using the ARToolkit tracking library. We found that the native Symbian AR application (based on the Symbian version of ARToolKit) ran 6 times faster than our J2ME application (6FPS VS 1FPS). However, porting the Symbian application between different models of mobile phones will take approximately one to three months, while porting J2ME applications between different models of mobile phones will take approximately one or two days. The performance of the native Symbian applications is much better than the J2ME applications, but the portability of Symbian applications is much worse. Figure 5-8 shows the Symbian application and J2ME application.



a) Output of ARToolkit application on 6680 b) Output of our application on 6680.

Figure 5-8 shows the Symbian application and J2ME application

5.2.4 Test Pose Estimation Accuracy

Another experiment was run to test the pose estimation accuracy in our application. We rotated the marker at different angles and compared it with the value from our program. The width of the marker used in this experiment was 60mm. First, we fixed the camera on the top of the marker. The distance from the camera of mobile phone to the marker was 120mm. This was the shortest distance which our application can detect a marker. A protractor was used to measure the marker angle. (Refer to Figure 5-9).

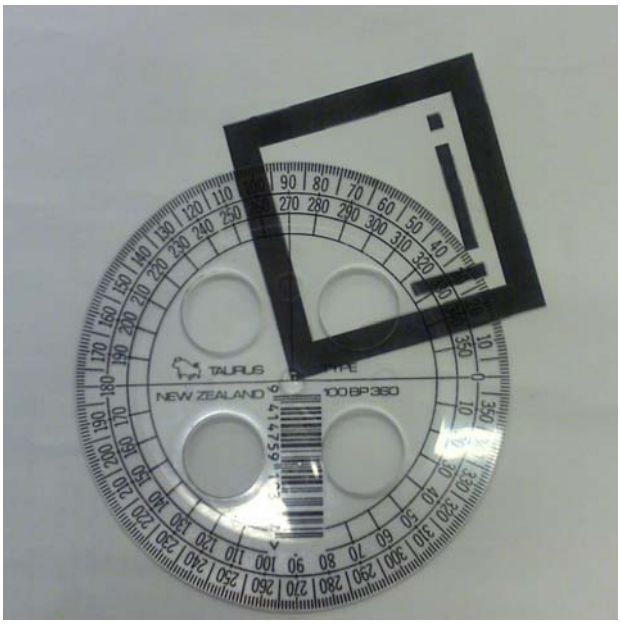


Figure 5-9 Testing the different angle.

We rotated the marker every 10 degrees, and took a picture each time. We took 36 pictures all together and placed these pictures in our application simulator to validate the output data of the angles. If the application output the same degree of the angle which we tested, we define this is correct. We tested 36 different angles, and noticed that our application is quite accurate. There were five angles that did not show up correctly from

the 36 angles. The accuracy of the detection function in our application is 86% (this value was calculated by, 36 subtracts five then divided by 36). Next, we added 60mm to the distance between the camera and the desk to test the accuracy in the above method. The accuracy of the angle rotation dropped down to 78%, and 28 of 36 angles were correct. We increased the distance continuously until the application could not recognize the marker, which occurred at 300mm. The detection of accuracy of our application is displayed on Figure 5-10. We can see when the distance increases the detection accuracy decreases dramatically.

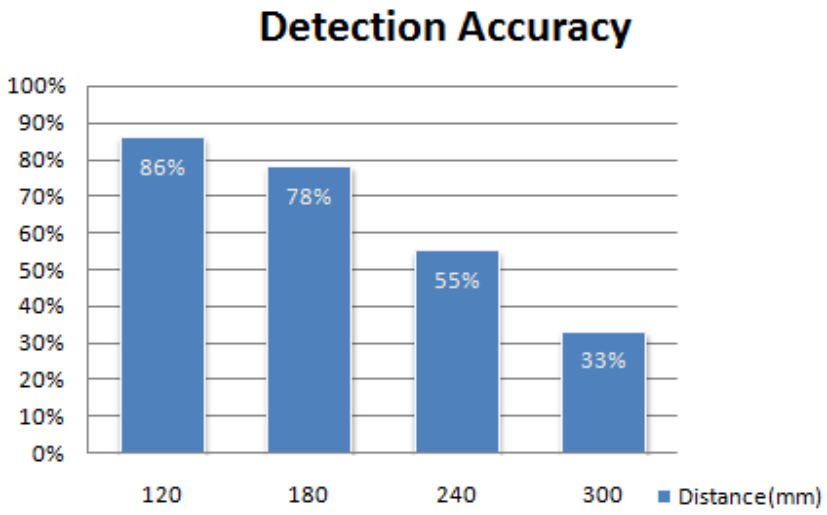


Figure 5-10 The detection accuracy with increase in distance.

In addition, we chose four different angles in each different distance (120, 180, 240, 300mm) to identify the accuracy variations between these angles. The angles that we randomly chose were 36, 67, 103 and 127. The results from this experiment are displayed on Figure 5-11. From this figure, we can see that error in the detected angle increases with the distance between the marker and camera.

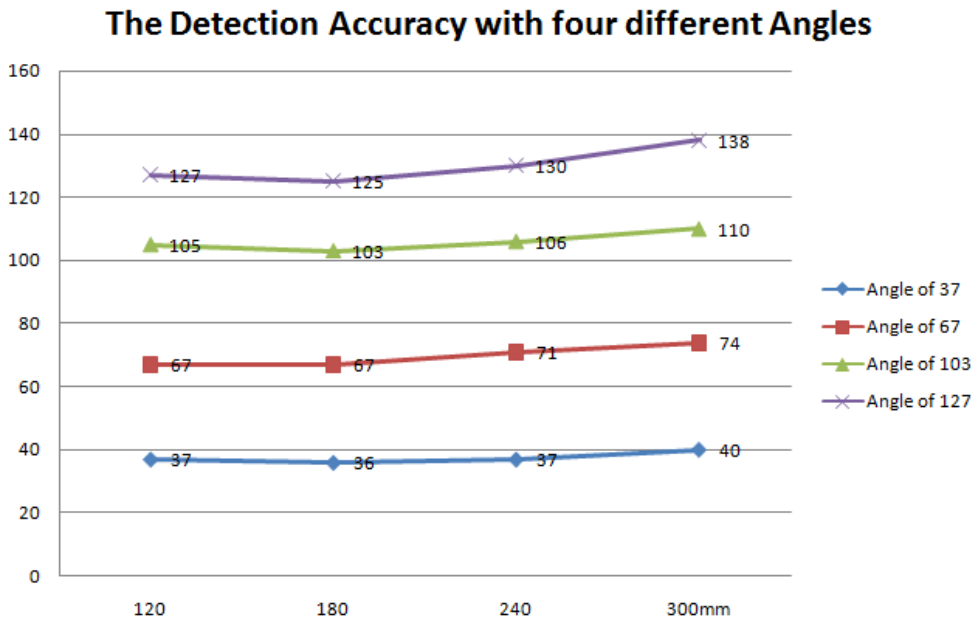


Figure 5-11 The detection accuracy with four different angles.

5.3 Motion Estimation Application

Motion in video sequences can be categorized by 3 methods: Camera movement, the movement of objects within a frame, or the movement of both camera and objects. To test our motion estimation code, we chose the second method (movement of objects within a frame). In the test, a mobile camera was placed in a fixed position. The performance of the motion estimation application was evaluated using a range of complex motions from slow motion to a very fast motion, translation and zooming.

During the experiments, all sequences were first converted to a gray level intensity frame. The block size dimensions were set at 16*16 and d = ±7. Thus, within each 16*16 block,

a maximum of $(2d+1)^2 = 225$ checking points were used. The performance of the TSS and FS algorithms was quantitatively evaluated using the following two measures:

- the efficiency of the algorithm (FPS, and time taken during processing)
- the average number of search points for computational complexity

The search points indicate how many pixels are searched and the more pixels that have been searched by the algorithm, the greater the likelihood of accurate results. However, this also increases the time taken for the search. The results are summarized in Table 5-4.

BMA	SP	FPS/Elapsed time of each frame
FS	200	0/1200ms
TSS	30	0/19400ms

Table 5-4: shows the experimental results for the average search points and FPS comparisons for the 25 frames of our mobile camera.

We also tested motion estimation algorithms in different phones. Table 5-5 shows the results. From the comparisons it is obvious that the FS algorithm has better performance on search point but the time spent in FS is a lot more than the time TSS algorithm used. The TSS algorithm is therefore more suitable for the J2ME library.

Mode:	CPU (MHZ)	Motion Estimation (ms)	
		FS	TSS
Nokia 6680	220	17490-19400	1200-1400
Nokia 6630	104	16000-16200	1240-1400
Nokia N72	220	17200-18900	1130-1380

Nokia N95	332	12600-14200	8000-9000
Motorola E680i	200	13790-19170	1200-1400
SonyEricsson W810i	No	19000-21040	1900-2100
SonyEricsson W530	No	failed	failed

Table 5-5: Time required for motion estimation in different phones.

5.4 Edge Detection Application

We implement an edge detection application by using the Sobel and Canny algorithms, described in Section 4.10. Two experiments were used to evaluate the performance of our edge detection application. The first one used pictures taken in different environments to compare the performance of the two algorithms. The second experiment compared the performance of our Sobel edge detection application with a Sobel edge detection application [57] implemented on the Symbian C++ platform.

5.4.1 Compare the Performance of Sobel and Canny Algorithms

In order to evaluate the performance of the Sobel and Canny algorithms, the following two measures were considered:

- the efficiency of the algorithm (elapsed time for picture processing)
- threshold value of the algorithms

Table 5-6 shows the processing time for each edge detector and the corresponding picture for different threshold values. The visual comparison of these images can lead us to the

subjective evaluation of the performances of selected edge detectors. Applying Canny and Sobel operators to a noisy image shows that with different threshold values, Canny exhibits better performance but requires more computations (because of the initial Gaussian smoothing, then computing the gradient). In Sobel, the processing time is satisfactory but the algorithm has less ability to prevent noise.

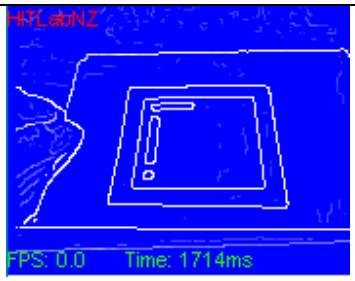
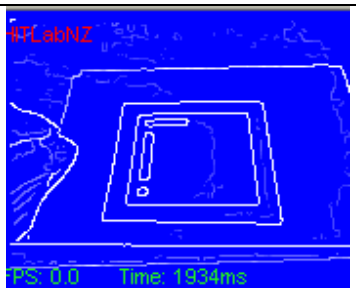
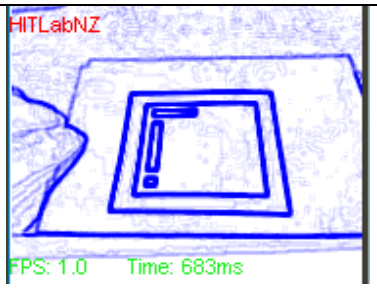
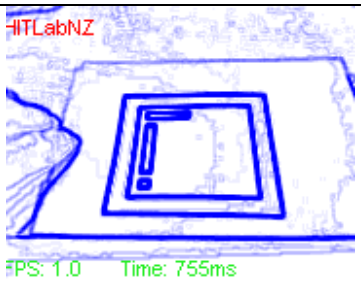
Algorithm	**Threshold Value**		**Processing Time**
	(r * 30 + g * 59 + b * 11) / 100;	**(r + g) / 2;**	
Canny			1600-2000
Sobel			680- 800

Table 5-6: Compare the Performance of Sobel and Canny algorithms

5.4.2 Comparing J2ME edge detection with a Symbian C++ application.

For a second experiment, we compared our Sobel detection algorithm with the existing Sobel edge library which was developed on the Symbian C++ platform. This Symbian

C++ application is developed by Charles Han [57]. The core code of the Sobel algorithm is the same. We used the Nokia N72 mobile phone to test the two applications. Figure 5-12 shows the two applications run on the Nokia N72.

The experiment showed that that the Symbian application runs much faster than the J2ME version (230ms and FPS = 4 for the Symbian application, compared with 860ms and FPS = 1 for the J2ME version). Because the core codes of the algorithms are same, we did not find any difference in the detection accuracy of the platforms.

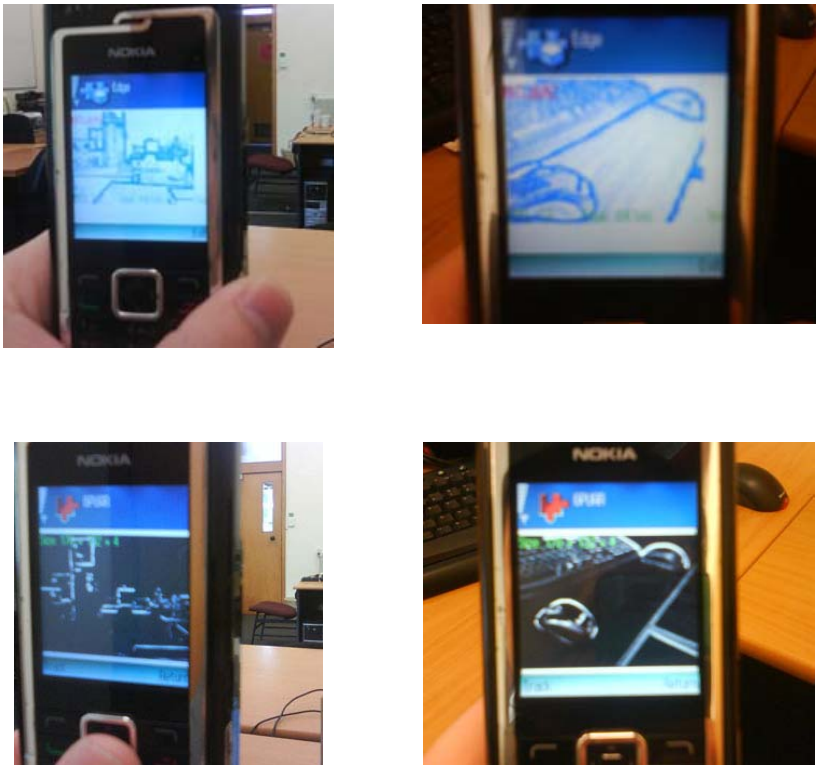


Figure 5-12 Implementations of the Sobel edge detection algorithm. J2ME application on Nokia 6680 (top) and Symbian C++ application on Nokia N72 (bottom).

5.5 Problems

The problem in our application is still the long processing time. We tried using different methods to increase the FPS rate: decreasing resolution, using fixed point library and optimising the code in our application but the improvement was not substantial.

5.5.1 Long Processing Time

We tried to improve our application performance by using different methods such as decreasing the camera image resolution (described in section 5.2.1), optimising the code, and using a fixed point library.

Optimising Code

We tried optimising the code in our application by following the rules recommended in [58] which include:

- **Reducing the use of Exceptions:** Testing Exceptions is a time consuming process in J2ME, so we only use Exceptions in our code when we have no other options.
- **Using Threads:** We try to use different threads in J2ME application. However, J2ME already provides different specific thread to do specific tasks, such as taking pictures, rendering graphics and creating 3D objects. So, introducing a new thread does not improve performance a lot.
- **Reusing objects:** J2ME takes a long time to create an object, so we tried to update the fields of an old object and reusing it rather than creating a new object.

- **Using high level graphics primitives:** For example, in J2ME, it is much faster to call drawPolygon() on a bunch of points than looping with drawLine(), so we try to use high level primitives in Graphics.

- **Avoiding expensive data structures:** We tried to use simpler data structures instead of the more complex ones. For example, we would use several one-dimensional arrays instead of a two-dimensional array.

Fixed Point Library (FPL)

Another method we used to increase performance was to develop a fixed point library.In J2ME, there are two key components: Connected Limited Device Configuration (CLDC) [59] and Mobile Information Device Profile (MIDP) [60]. Together they form the J2ME application environment. Currently, there are two versions of CLDC and MIDP. Floating point values are not supported in CLDC1.0/MIDP1.0 but are supported from CLDC1.1/MIDP2.0 onwards. M3G requires floating point support, so we used CLDC1.1/MIDP2.0 as our application environment. Although our application was developed in CLDC1.1, floating point arithmetic is computationally expensive. We trailed a Fixed Point Library [43] as a potential method to increase the efficiency of the algorithms. During the experiments, the Fixed Point Library did not really improve the efficiency of algorithms. We broke down each process to evaluate the effect of the FPL on the original algorithm. Table 5-7 shows the run time comparisons between the fixed point and floating point methods. The fixed point library increased the time for second order moments, homography calculation and pose estimation, so FPL is not a promising solution.

Algorithms:	Time (ms)	
	Not Using FPL	**Using FPL**
Take picture in camera	237- 260	247-267
Threshold	175-190	165-192
Contour Detection	65-80	65-80
Rectangle Fitting	65-83	65-81
Second Order Moments	**26-40**	**30-80**
Homography Calculation	**20-43**	**40-80**
Pose Estimation Algorithm	**32-40**	**60-72**
Using M3G	20-30	20-32

Table 5-7: Runtime comparisons between floating and fixed point libraries

5.5.2 Camera Calibration Decreases the Portability of Our Application

For accurate pose estimation, our application needs the camera parameters of the camera in the phone used. We can use the MATLAB camera calibration toolbox [61] to get the camera parameters. For this calibration method the user takes multiple pictures of a checker board, then the toolbox loads the images. The whole procedure typically takes five minutes. This acquisition procedure has decreased the portability of our application. It is time consuming to calibrate cameras for different phones. In order to further explore the effectiveness of camera calibration, we used the same calibration parameter (for the Nokia 6680 mobile phone) for several different phones. The camera calibration parameters did not significantly affect our application in other phones from the same

vendor. However, the pose estimation was substantially different on the phones from Motorola and SonyEricsson. One possible cause could be the different phone manufacturers use different standard lenses, so that camera parameters found for one manufacturers phones can not be used for a different phone manufacturer. So, the camera calibration is one factor that would affect the portability of our application.

6 Conclusions

In this research, we focused on the design, development and implementation of computer vision algorithms using J2ME for mobile phone applications. We reviewed several papers that show both the importance as well as the complexity of this project. In this thesis, we discussed the main functions and the structure of our computer vision library and showed the implementation of each algorithm in the library. Finally, we conducted several experiments to evaluate the compatibility, portability and efficiency of our applications.

The experiments show that one of the biggest problems in mobile phone computer vision on the J2ME platform is the fact that different mobile phones support different implementations of J2ME. Although most of mobile phones on the market claim to support Java, the Low-level Design layer for supporting the Java API is different and it is hard to improve the performance of these standards. A few examples are the ‘click’ sound, resolution adjustment and the time to take a picture.

From this project, we can see that mobile augmented reality and computer vision using J2ME are still in the early stages of development. If J2ME can be used in low level design and the Java API can be standardized across different phones, it will be easier to improve the performance of augmented reality applications developed in J2ME platform.

Future developments such as MIDP 3.0 (JSR 271) and M3G 2.0 (JSR 297), and improvements in mobile phone hardware (such as faster processors and graphics chips) will also be important.

In the future, in order to carry out the development of AR applications on J2ME platform, one of the possible directions will be to used Windows Mobile phones to carry out research. Most mobile phones which run the Windows Mobile operating system support the J2ME platform. If we can familiarize ourselves with the Java API used in Windows Mobile phones, we can port our application to these phones. Windows Mobile phones are more powerful than the mobile phones which we used in our research.

Another possible direction for further research would be to explore faster computer vision algorithms for the J2ME platform. The algorithms of second order moments, pose estimation and homogeneous transformation could be improved to increase performance. We hope in the future, we can see more research in Augmented Reality applications for the J2ME platform, and our J2ME computer vision library can be seen as a stepping stone for the future research.

As part of this research, we had submitted the following paper to the IVCNZ08 conference, which has now been accepted for oral presentation:
Gu, J., Mukundan, R., Billinghurst, M., "**Developing Mobile Phone AR Applications Using J2ME**" , Accepted for oral presentation, Image and Vision Computing - IVCNZ 2008.

References

[1] GARTNER REASEARCH
http://www.gartner.com/it/products/research/research_services.jsp

[2] Wikipedia --- Augmented Reality
http://en.wikipedia.org/wiki/Augmented_reality

[3] Windows Mobile
http://www.microsoft.com/windowsmobile/

[4] Palm OS
http://www.palm.com/us/support/downloads/

[5] Symbian OS
http://www.symbian.com/symbianos/

[6] Macromedia's Flash Lite
http://www.adobe.com/products/flashlite/

[7] DoJa
http://www.doja-developer.net/downloads/

[8] J2ME
http://java.sun.com/javame/

[9] BREW
http://brew.qualcomm.com/brew/en/

[10] Execution Environment
http://developer.in-fusio.com/exen/

[11] Chaisatien, P. and K. Akahori (2007). “A Pilot Study on 3G Mobile Phone and Two Dimension Barcode in Classroom Communication and Support System”. Advanced Learning Technologies, 2007. ICALT 2007.

[12] High Energy Magic Ltd. The spotcode framework. Available for download from: http://www.highenergymagic.com/

[13] Semacode website
http://semacode.org/

[14] Rekimoto, J., Ayatsuka, Y. (2000) 'CyberCode: Designing Augmented Reality Environments with Visual Tags',
http://www.cs.sony.co.jp/person/rekimoto/papers/dare2000.pdf.

[15] Lopez de Ipina, D., Mendonca, P., Hopper, A.: TRIP: A low-cost vision-based location system for ubiquitous computing. Personal and Ubiquitous Computing 6 (2002) 206--219 http://citeseer.ist.psu.edu/lopezdeipina02trip.html

[16] “Information Technology – International Symbology Specification – Data Matrix”. International Organization for Standardization/International Electrotechnical Commission / 01-May-2000 / 95 pages

[17] Eleanor, T., S. Richard, et al. (2007). "Interacting with mobile services: an evaluation of camera-phones and visual tags." Personal Ubiquitous Comput. 11(2): 97-106.

[18] PullFace

http://dotsis.com/mobile_phone/showthread.php?p=331163

[19] Rohs., M “Real-World Interaction with Camera-Phones”. 2nd International Symposium on Ubiquitous Computing Systems (UCS 2004). Lecture Notes in Computer Science (LNCS) No. 3598, Springer-Verlag, pp. 74-89, Tokyo, Japan, November 2004

[20] Rohs, M.: Marker-Based Embodied Interaction for Handheld Augmented Reality Games, In Proceedings of the 3rd International Workshop on Pervasive Gaming Applications (PerGames) at PERVASIVE 2006, 2006, Ireland

[21] Mosquito Hunt.

http://w4.siemens.de/en2/html/press/newsdesk_archive/2003/foe03111.html

[22] Marble Revolution Website.

http://www.bitside.com/entertainment/MOBILE%20GAMES/Marble

[23] Hakkarainen, M., Woodward, C., SymBall: camera driven table tennis for mobile phones, Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology, p.391-392, June 15-17, 2005, Valencia, Spain

[24] Volker, P., R. Christian, et al. (2004). Foot-based mobile interaction with games. Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology. Singapore, ACM Press.

[25] Tiny Motion Library

http://guir.berkeley.edu/projects/tinymotion/

[26] Jingtao, W. and C. John (2006). TinyMotion: camera phone based interaction methods. CHI '06 extended abstracts on Human factors in computing systems. Montrbec, Canada, ACM Press.

[27] ARToolkit

http://www.hitl.washington.edu/artoolkit/

[28] Wagner, D., Schmalstieg, D., First Steps Towards Handheld Augmented Reality, 7th International Conference on Wearable Computers (ISWC 2003), White Plains, NY, 2003

[29] Moehring, M., Lessig, C. and Bimber, O., Video See-Through AR on Consumer Cell Phones, International Symposium on Augmented and Mixed Reality (ISMAR'04), pp. 252-253, 2004

[30] Henrysson, A., M. Billinghurst & M. Ollila, (2005), “Face to Face Collaborative AR on Mobile Phones”, In Proceedings of the Fourth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR’05), 2005.

[31] Wagner, D.”Handheld Augmented Reality” PHD thesis October 2007, Graz University of Technology.

[32] Nokia Research Centre
http://research.nokia.com/

[33] Nokia Vision Library (NCV)
http://research.nokia.com/research/projects/nokiacv/

[34] Phone Vision Library (PVL)
http://www.newlc.com/Phone-Vision-Library-release-1.html

[35] Paul, F., ckler, et al. (2005). PhoneGuide: museum guidance supported by on-device object recognition on mobile phones. Proceedings of the 4th international conference on Mobile and ubiquitous multimedia. Christchurch, New Zealand, ACM Press.

[36] Groeneweg, N., Groot, B., Halma, A., Quiroga, B. (2006). "DOAS 2006 Project Article: Mobile Landmark Recognition."

[37] Gavilan Ruz, D. (2005). "Mobile Image Categorization and Retrieval using Color Blobs."

[38] Rekimoto, J., Ayatsuka, Y. (2000) 'CyberCode: Designing Augmented Reality Environments with Visual Tags', http://www.cs.sony.co.jp/person/rekimoto/papers/dare2000.pdf.

[39] Chaisatien, P. and K. Akahori (2007). “A Pilot Study on 3G Mobile Phone and Two Dimension Barcode in Classroom Communication and Support System”. Advanced Learning Technologies, 2007. ICALT 2007. Seventh IEEE International Conference on. on Advanced Learning Technologies.

[40] Föckler, P., Zeidler, T., Brombach, B., Bruns, E., and Bimber, O. PhoneGuide: Museum Guidance Supported by On-Device Object Recognition on Mobile Phones In proceedings of International Conference on Mobile and Ubiquitous Computing (MUM'05), 2005

[41] Pulli, K., T. Aarnio, et al. (2005). "Designing graphics programming interfaces for mobile devices." Computer Graphics and Applications, IEEE 25(6): 66-75.

[42] MMAPI
http://java.sun.com/products/mmapi/

[43] Fixed Point Library
http://mywebpages.comcast.net/ohommes/MathFP

[44] Li, S., Knudsen, J., “Beginning J2ME: From Novice to Professional” Apress; 3rd edition, 2005.

[45] Pierre D.Wellner: “Adaptive Thresholding for the DigitalDesk”. Technical Report EPC-93-110, Rank Xerox Research Centre, Cambridge, UK, 1993.

[46] Veltkamp, R. C. and Hagedoorn, M.: State of the Art in Shape Matching. Principles of Visual Information Retrieval, Michael S. Lew (Ed.), Series in Advances in Pattern Recognition, Springer, 2001.

[47] Mukundan, R& Ramakrishnan, K R. *Moments Functions In Image Analysis – Theory and Applications.* Published by World Scientific.

[48] Hartley, R ., A.Zisserman, , Multiple View Geometry in Computer Vision (2nd Edition),

Cambridge Pres,s 2003.

[49] Kanopoulos N., Yasanthavada N., and Baker R., "Design of an Image Edge Detection Filter Using the Sobel Operator", IEEE Journal of Solid State Circuits, vol. 23, NO. 2, April 1988, pp.358-367.

[50] Sobel Edge Dector

http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

[51] Canny Edge Detection Tutorial

http://www.pages.drexel.edu/~weg22/can_tut.html

[52] Jong-Nam, K. SeongChul, B. and ByungHa, A. "Full search based fast block matching algorithm with efficient matching order in motion estimation," IEICE Transactions on Communications, vol. E86-B, pp. 1191-5, 2003.

[53] Yeong-Kang, L. "A memory efficient motion estimator for three step search block-matching algorithm,. IEEE Transactions on Consumer Electronics, vol.47, no.3, Aug. 2001, pp.644-51. USA.

[54] Search Algorithms for Block-Matching in Motion Estimation

http://www.ece.cmu.edu/~ee899/project/deepak_mid.htm

[55] 3D Maze Game

http://www.forum.nokia.com/document/Java_ME_Developers_Library_v2/

[56] Anders Henrysson

http://www.hitlabnz.org/wiki/Anders_Henrysson

[57] Charles Han

http://www.hitlabnz.org/wiki/Charles_Han

[58] Optimizing Java for Speed

http://www.cs.cmu.edu/~jch/java/speed.html

[59] CLDC

http://java.sun.com/products/cldc/

[60] MIDP

http://java.sun.com/products/midp/

[61] MATLAB camera calibration toolbox

http://www.vision.caltech.edu/bouguetj/calib_doc/

[62] Xu, L., D. David, et al. (2006). Imaging as an alternative data channel for camera phones. Proceedings of the 5th international conference on Mobile and ubiquitous multimedia. Stanford, California, ACM Press.

[63] Hiroyuki, T., Y. Kayo, et al. (2006). Open experiments of mobile sightseeing support systems with shared virtual worlds. Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology. Hollywood, California, ACM Press

[64] Rudolph, P. D. and D. Richard (2005). Mixed-dimension interaction in virtual environments. Proceedings of the ACM symposium on Virtual reality software and technology. Monterey, CA, USA, ACM Press.

[65] Kai, K., T. Marko, et al. (2006). Toolkit for user-created augmented reality games. Proceedings of the 5th international conference on Mobile and ubiquitous multimedia. Stanford, California, ACM Press.

[66] Wagner, D., Schmalstieg D. , Billinghurst M. (2006). "Handheld AR for Collaborative Edutainment."

[67] Sam, B., B. Mark, et al. (2005). User experiences with mobile phone camera game interfaces. Proceedings of the 4th international conference on Mobile and ubiquitous multimedia. Christchurch, New Zealand, ACM Press.

[68] Hirokazu, K. and B. Mark (1999). Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing System. Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality, IEEE Computer Society.

[69] Çapın, T., Haro, A., Setlur, V., Wilkinson, S. (2006). "Camera-Based Virtual Environment Interaction on Mobile Devices."

[70] Billinghurst, M., Kato, H., Weghorst, S. and Furness, T. A. A Mixed Reality 3D Conferencing Application. Technical Report R-99-1 Seattle: Human Interface Technology Laboratory, University of Washington, 1999

[71] Monkey Physics Game Engine for J2ME
http://www.mamlambo.com/mdn/archives/000312.html

[72] FREE 3D ENGINE DEMO (JAVA / J2ME) FOR MOBILES
http://rzr.online.fr/docs/java/00index.htm

[73] Wells, M. “J2ME Game Programming.” Thomson Course Technology, 2004.

[74] Goyal,V. “Pro Java ME MMAPI: Mobile Media API for Java Micro Edition.” Apress,. 2006.

[75] Höfele, C. “Mobile 3D Graphics: Learning 3D Graphics with the Java Micro Edition” Course Technology PTR 2007

[76] Nixon, M. S., Aguado, A, Feature Extraction and Image Processing, Butterworth-Heinmann, pp 114, (2002).

[77] Sorwar, G., Murshed, M. and Dooley, L. (2004). Fast Block-Based True Motion Estimation Using Distance Dependent Thresholds. Journal of Research and Practice in Information Technology, pp. 157–158.

[78] Phadtare, M. Motion estimation techniques in video processing, Electronic Engineering Times India, August 2007.

[79] Heckbert, P.S., Fundamentals of Texture Mapping and Image Warping MS thesis, Dept. of Electrical Eng. and Computer Science, Univ. of California, Berkeley, June 1989.

[80] Soongsathitanon, S. Woom, W.L. Dlay, S.S. (2005), Fast search algorithms for video coding using orthogonal logarithmic search algorithm, Sch. of Electr., Electron. & Comput. Eng., Newcastle upon Tyne Univ., UK,

Appendix A. Adaptive Threshold Method (J2ME)

```
//These code follow the algorithm which described in [19]
public int[] doThresholding(int[] im1_d ) {

        int s = w / 4; // s: number of pixels in the moving average (w = image width)
        int t = 30;    // t: percentage of brightness range considered white

        final int S = 9; // integer shift, needed to avoid floating point operations
        final int power2S = 1 << S;

        // for speedup: multiply all values by 2^s, and use integers instead of doubles

        int factor = power2S * (100-t) / (100*s); // multiplicand for threshold

        int gn = 127 * s; // initial value of the moving average (127 = average gray value)
        int q = power2S - power2S / s; // constant needed for average computation

        // store previous scanline of gn-values required to avoid artifacts in each other line
        int [] prev_gn = new int[w];

        // initalize prev_gn with initial gn value
        for(int i =0; i<w; i++){
                prev_gn[i] = gn;
        }


        int pn, hn;
        int x, y;

        // used to store the label equivalences (index 0 unused)
        equals = new int [maxSplinter + 1];
        currentCluster = 0;


        // run through all pixels in a zig-zag fashion
        for (y = 0; y < h; y++) {
                for (x = 0; x < w; x++) { // left-to-right

                        pn = imageToGrayScale(im1_d[y*w+x]);

                        gn = ((gn*q)>>S) + pn; // update average

                        // average of current pixel and the one above it
                        hn = (gn + prev_gn[x]) >> 1;
                        prev_gn[x] = gn; // update previous scanline memory
```

```
                        if (pn < (hn*factor)>>S ) {
                                int top = (y>0) ? p[y-1][x] : BACKGROUND;
                                int left = (x>0) ? p[y][x-1] : BACKGROUND;
                                processForegroundPixel(x, y, top, left);
                        } else {
                                p[y][x] = BACKGROUND;
                                qp[y*w+x] = BACKGROUND;
                        }
                }

                y++;
                if (y == h) break;

                for (x = w-1; x >= 0; x--) { // right-to-left
                        pn = imageToGrayScale(im1_d[y*w+x]);
                        gn = ((gn*q)>>S) + pn; // update average
                        // average of current pixel and the one above it
                        hn = (gn + prev_gn[x]) >> 1;
                        prev_gn[x] = gn; // update previous scanline memory

                        if (pn < (hn*factor)>>S) {
                                int top = (y>0) ? p[y-1][x] : BACKGROUND;
                                int right = (x<w-1) ? p[y][x+1] : BACKGROUND;
                                processForegroundPixel(x, y, top, right);
                        } else {
                                p[y][x] = BACKGROUND;
                                qp[y*w+x] = BACKGROUND;
                        }
                }

        }


        return qp;

        }
```

Appendix B. Second Order Moments (J2ME)

```
//These code follow the algorithm which described in [19]
public Vector findClusters() {

		int [] label = new int [currentCluster + 1]; // index 0 unused
		int currentLabel = 0;



		for (int i = 1; i <= currentCluster; i++) { // index 0 unused
			if (label[i] == 0) { // cluster i has no label yet
				currentLabel++;
				// run through equality-chain and set current label
				int j = i;
				while (label[j] == 0) {
					label[j] = currentLabel;
					j = equals[j];
				}
			}
		}

		 int [] cluster_size = new int[currentLabel+1];
		 long [] cluster_x_sum = new long[currentLabel+1];  // for center calculation
		 long [] cluster_y_sum = new long[currentLabel+1];
		 int [] cluster_min_x = new int[currentLabel+1];  // bounding boxes
		 int [] cluster_max_x = new int[currentLabel+1];
		 int [] cluster_min_y = new int[currentLabel+1];
		 int [] cluster_max_y = new int[currentLabel+1];



		 for (int ko= 1; ko<= currentLabel; ko++){
			cluster_min_x[ko] = d_w-1;
			cluster_min_y[ko] = d_h-1;
		 }



		 for (int y = 0; y < d_h; y++) {
				for (int x = 0; x < d_w; x++) {
					int i = p[y][x]; // p contains unresolved cluster labels
					if (i == BACKGROUND) continue;
					i = label[i]; // use new label as index
					cluster_size[i]++;
					cluster_x_sum[i] += x;
					cluster_y_sum[i] += y;
					if (cluster_min_x[i] > x) cluster_min_x[i] = x;
```

```
                                if (cluster_max_x[i] < x) cluster_max_x[i] = x;
                                if (cluster_min_y[i] > y) cluster_min_y[i] = y;
                                if (cluster_max_y[i] < y) cluster_max_y[i] = y;
                                // update p with the correct (resolved) label
                                p[y][x] = i;
                        }
                }
        Vector clusters = new Vector(currentLabel);

        for (int i = 1; i <= currentLabel; i++) {

                int c_x = (int) Math.ceil(cluster_x_sum[i] / (double)cluster_size[i]);
                int c_y = (int) Math.ceil(cluster_y_sum[i] / (double)cluster_size[i]);

                // calculate second-order moments of cluster i

                long mx = 0;
                long my = 0;
                long mxy = 0;

                //iterate over bounding box
                for (int x = cluster_min_x[i]; x <= cluster_max_x[i]; x++) {
                        for (int y = cluster_min_y[i]; y <= cluster_max_y[i]; y++) {
                                if (p[y][x] == i) { // if point belongs to cluster
                                        int xdiff = x - c_x;
                                        int ydiff = y - c_y;
                                        mx += xdiff * xdiff;
                                        my += ydiff * ydiff;
                                        mxy += xdiff * ydiff;
                                }
                        }
                }
                // divide by size
                double m_x = mx / (double)cluster_size[i];
                double m_y = my / (double)cluster_size[i];
                double m_xy = mxy / (double)cluster_size[i];

                // calculate parameters of ellipse

                double factor = (4 * m_x * m_y - m_xy * m_xy);



                if (factor == 0.0)
                else factor = 1.0 / factor;
                double d =  m_y  * factor;
                double e = -m_xy * factor;
                double f =  m_x  * factor;
                // calculate (absolute) eigenvalues
                double root = Math.sqrt((d-f)*(d-f) + 4*e*e);
                double lambda1 = Math.abs((d + f + root) / 2);
                double lambda2 = Math.abs((d + f - root) / 2);
                // calculate eccentricity
```

```
                                double ratio = (lambda1<lambda2) ? lambda1/lambda2 : lambda2/lambda1;
                                // calculate direction
                                mathequations mathe = new mathequations();
                                double alpha =  0.5 * mathe.atan2(2*e, d-f);

                                ClusterInfo ci = new ClusterInfo(i, c_x, c_y, cluster_min_x[i], cluster_max_x[i],
cluster_min_y[i], cluster_max_y[i], ratio, alpha, cluster_size[i], lambda1, lambda2);
                                clusters.addElement(ci);

                        }
                 return clusters;
                }
```

Appendix C. Pose Estimation Algorithm (J2ME)

```
//These code is original from STBTracker. We translate it to J2ME. Details is described in Section 4.

	public class PoseEstimation {

		Marker_PIMPL marker = new Marker_PIMPL();

		Vec3[] nPoints3D = new Vec3[4];
		Vec2[] nPoints2D =  new Vec2[4];

		int [][] screenCoordi;
		float[] correctrotation;

		Matrix32 homography;
		Matrix32 camMatrix;
		Matrix32 invCamMat;
		Matrix32 pose;

		int  d_w, d_h;

		double cpara[];

		public PoseEstimation(float[] homo, double[] cam, int[][] coordi2D, int w, int h) {
			cpara = cam;
			screenCoordi = coordi2D;
			d_w = w;
			d_h = h;
			correctrotation = homo;
			//homography = new Matrix32(-homo[0],-
homo[1],homo[2],homo[3],homo[4],homo[5],homo[6],homo[7],homo[8]);
			camMatrix = new Matrix32(3,4);
			invCamMat = new Matrix32(3,4);
			pose = new Matrix32(3,4);

		}


		public Marker_PIMPL poseEstimation(){

	IntPoint lt = BarrelDistortion.correctDistorted(screenCoordi[0][0], screenCoordi[0][1], d_w, d_h);

	IntPoint rt = BarrelDistortion.correctDistorted(screenCoordi[1][0], screenCoordi[1][1], d_w, d_h);

IntPoint rb = BarrelDistortion.correctDistorted(screenCoordi[2][0], screenCoordi[2][1], d_w, d_h);

IntPoint lb = BarrelDistortion.correctDistorted(screenCoordi[3][0], screenCoordi[3][1], d_w, d_h);

			Warper warper = new Warper(lt.x, lt.y, rt.x, rt.y, rb.x, rb.y, lb.x, lb.y, false);
```

```
int rotation = getCodeRotation(warper);

//System.out.println("rotation:"+rotation);

float ha = (float)warper.geta();
float hb = (float)warper.getb();
float hc = (float)warper.getc();

if (hc >=0){
        if(correctrotation[2]<0)
                hc = -correctrotation[2];
        else
                hc = correctrotation[2];
}
else
        if(correctrotation[2]<0)
                hc = correctrotation[2];
        else
                hc = -correctrotation[2];

//float hc = (float)-correctrotation[2];
float hd = (float)warper.getd();
float he = (float)warper.gete();
float hf = (float)warper.getf();

if (hf >=0){
        if(correctrotation[5]<0)
                hf = -correctrotation[5];
        else
                hf = correctrotation[5];
}
else
        if(correctrotation[5]<0)
                hf = correctrotation[5];
        else
                hf = -correctrotation[5];


//float hf = (float)-correctrotation[5];
float hg = (float)warper.getg();
float hh = (float)warper.geth();
float hi = (float)1;

homography = new Matrix32(-ha,-hb,hc,hd,he,hf,hg,hh,hi);


nPoints3D[0] = new Vec3((float)-30,(float)-30,0);
nPoints3D[1] = new Vec3((float)30,(float)-30,0);
nPoints3D[2] = new Vec3((float)30,(float)30,0);
nPoints3D[3] = new Vec3((float)-30,(float)30,0);
```

```
                nPoints2D[0] = new Vec2(screenCoordi[0][0], screenCoordi[0][1]);
                nPoints2D[1] = new Vec2(screenCoordi[1][0], screenCoordi[1][1]);
                nPoints2D[2] = new Vec2(screenCoordi[2][0], screenCoordi[2][1]);
                nPoints2D[3] = new Vec2(screenCoordi[3][0], screenCoordi[3][1]);


                float x[] = new float[12];

                for( int s = 0; s<12; s++ ){
                        x[s] = (float)cpara[s];
                }
                camMatrix.setmat1d(x);
                float[][] c2 = camMatrix.getmat2d();

                invCamMat = new Matrix32(3,3);

                for(int r=0; r<3; r++)
                        for(int c=0; c<3; c++)
                                invCamMat.set2d(r,c,c2[r][c]);
                invCamMat.inverse();

                marker.setHomography(homography);
                marker.setRroatation(0);
                marker.setcornerPoints3dF(nPoints3D);
                marker.setcornerPoints2dF(nPoints2D);

         getPoseFromHomography(pose, marker.homographyF, invCamMat, marker.getSize());

                fixInitialPose(pose, marker,camMatrix);

                marker.setposeF(pose);
                marker.argConvGlpara(pose.getmat2d());

                return marker;
        }
        /*pose estimation function*/

        void fixInitialPose(Matrix32 nPose, Marker_PIMPL nMarker, Matrix32 nCamMatrix)
        {
                Vec2 points2D[] = new Vec2[4];
                Vec3 points3D[] = new Vec3[4];
                Matrix32 projMat = new Matrix32();

                points3D = nMarker.getVec3();
                projMat = buildProjectionMatrix(nPose, nCamMatrix);

                //int dir = 0;

                int dir = nMarker.getRoatation();
                //int minRot=nMarker.getRoatation();

                int minRot = 0;
```

```
		float minErr = -1.0f;
		float errs[] = new float[4];


		if(minRot==0)
			return;


		Matrix32 R = new Matrix32(3,3);
		Matrix32 dR = new Matrix32(3,3);
		Matrix32 oldR = new Matrix32(3,3);
		Vector32 T =  new Vector32(3);


		decomposePose(nPose, oldR,T);



		switch(minRot)
		{
		case 1:	// rotate 90?
			dR.makeRotationZ(1,0);
			R.mul(R,oldR,dR);
			break;
		case 2:
			dR.makeRotationZ(0,-1);	// rotate 180?
			R.mul(R,oldR,dR);
			break;
		case 3:	// rotate -90?
			dR.makeRotationZ(-1,0);
			R.mul(R,oldR,dR);
			break;
		}

		composePose(R,T, nPose);
	}

	void decomposePose(Matrix32 nPose, Matrix32 nRot, Vector32 nTrans)
	{
		for(int r=0;r<3;r++)
			for(int c=0;c<3;c++)
				nRot.set(r,c, nPose.get(r,c));
		for(int r=0;r<3;r++)
			nTrans.operatormiddlebracketvalue(r, nPose.get(r,3)) ;
	}


	void composePose(Matrix32 nRot, Vector32 nTrans, Matrix32 nPose)
	{
		for(int r=0;r<3;r++)
			for(int c=0;c<3;c++)
				nPose.set(r,c,nRot.get(r,c));
		for(int r=0;r<3;r++)
```

```
                        nPose.set(r,3,nTrans.operatormiddlebracket(r)) ;
                }

                public Matrix32 buildProjectionMatrix(Matrix32 nPose, Matrix32 nCamMatrix)
                {
                        //Float camMat34[3][4];
                        Matrix32  poseMat = new Matrix32(4,4);
                        Matrix32 projMat = new Matrix32();
                        Matrix32 camMat = new Matrix32();
                        camMat = nCamMatrix;
                        camMat.operatorMultiply(1.0f/64);


                        poseMat.makeIdent();
                        for(int r=0; r<3; r++)
                                for(int c=0; c<4; c++)
                                        poseMat.set(r,c, nPose.get(r,c));


                        projMat.mul(projMat,camMat,poseMat);

                        Matrix32  nMat = new Matrix32(3,4);
                        nMat.copyFromSubMatrix(projMat, 0, 0);
                        return nMat;

                }


                float computeReprojectionError(Vec3[] p3D, Vec2[] p2D, int nPoints, Matrix32 nMatrix)
                {
                        float fAbsErrSq = 0.0f;

                        for(int i=0; i<nPoints; i++)
                        {
                                Vec2 projected = new Vec2();
                                projectPoint(projected, p3D[i], nMatrix);
                                // compute deviation
                                float dx = p2D[i].getfirst() - projected.getfirst();
                                float dy = p2D[i].getsecond() - projected.getsecond();
                                // update absolute error
                                fAbsErrSq += dx*dx+dy*dy;
                        }
                        return fAbsErrSq;
                }

                void projectPoint(Vec2 p2D, Vec3 p3D, Matrix32 nMatrix)
                {
                        float[] mat = nMatrix.getmat1d();


p2D.setfirst(p3D.getfirst()*mat[0] + p3D.getsecond()*mat[1] + p3D.getthird()*mat[2] + mat[3]);
p2D.setsecond(p3D.getfirst()*mat[4] + p3D.getsecond()*mat[5] + p3D.getthird()*mat[6] + mat[7]);
float p2 = p3D.getfirst()*mat[8] + p3D.getsecond()*mat[9] + p3D.getthird()*mat[10] + mat[11];
```

```
                        p2 = 1.0f/p2;
                        p2D.setfirst(p2D.getfirst()*p2);
                        p2D.setsecond(p2D.getsecond()*p2);
                }

public boolean getPoseFromHomography(Matrix32 nPose, Matrix32 nHomography, Matrix32 nInvCameraMatrix, int nMarkerSize){

                        Matrix32 hsnMat, scaleMat, hsMat;

                        hsnMat = new Matrix32 ();
                        scaleMat = new Matrix32 ();
                        hsMat = new Matrix32 ();

                        Vec2 invMarkerSize = new Vec2(1.0f/nMarkerSize, 1.0f/nMarkerSize);

                        nPose.setSize(3,4);

                        // Create a scale matrix to normalize the homography
                        float diagonal[] = { invMarkerSize.getfirst(), invMarkerSize.getsecond(), 1 };


                        scaleMat.makeDiagonal(3, diagonal);
                        // Scale back the homography to marker size
                        hsMat.mul(hsMat,nHomography, scaleMat);
                        // Multiply inverse camera with scaled homography
                        hsnMat.mul(hsnMat, nInvCameraMatrix, hsMat);

                        // Calculate normalizing value for translation
                        float l0 = hsnMat.columnLength(0),
                                l1 = hsnMat.columnLength(1);
                        float lm = 2.0f/(l0+l1);
                        // Calculate translation from 3rd column
                        for(int i=0; i<3; i++)
                                nPose.set(i,3, hsnMat.get(i,2) * lm);


                        // If we found a solution behind the camera,
                        // then simple flip the position...
                        boolean swapped = nPose.get(2,3)<0;
                        if(swapped)
                                for(int i=0; i<3; i++)
                                        nPose.set(i,3, -nPose.get(i,3));


                        Vector32 vecX,vecY,vecZ;
                        vecX =  new Vector32();
                        vecY =  new Vector32();
                        vecZ =  new Vector32();


                        vecX.fromMatrixColumn(hsnMat, 0);
```

```
		vecX.operatorMultiply( 1.0f/l0);
		vecY.fromMatrixColumn(hsnMat, 1);
		vecY.operatorMultiply( 1.0f/l1);


		vecZ.cross(vecX,vecY);
		vecY.cross(vecZ,vecX);

		for(int i=0; i<3; i++)
			nPose.set(i,0, -vecY.operatormiddlebracket(i));

		for(int i=0; i<3; i++)
			nPose.set(i,1, -vecX.operatormiddlebracket(i));

		for(int i=0; i<3; i++)
			nPose.set(i,2, -vecZ.operatormiddlebracket(i));

		return true;
	}



	public IntPoint warp(double u, double v) {

		double a,b,c,d,e,f,g,h;
		a = correctrotation[0];
		b = correctrotation[1];
		c = correctrotation[2];
		d = correctrotation[3];
		e = correctrotation[4];
		f = correctrotation[5];
		g = correctrotation[6];
		h = correctrotation[7];

		u *= 0.1; v *= 0.1;

		double denom = g*u + h*v + 1;
		if (denom == 0.0) return new IntPoint(0,0);
		int x = (int)Math.ceil((a*u + b*v + c) / denom);
		int y = (int)Math.ceil((d*u + e*v + f) / denom);
		return new IntPoint(x,y);
	}


	public int getCodeRotation(Warper warper) {

		IntPoint o = warper.warp(0,0);
		IntPoint x = warper.warp(60,0);
		double alpha = 0.0;
		mathequations mathe = new mathequations();
		if (o.y == x.y && o.x < x.x) {
			alpha = 0.0;
		} else if (o.x == x.x) {
```

```
                    if (x.y < o.y) {
                        alpha = Math.PI / 2.0;
                    } else {
                        alpha = 3.0 * Math.PI / 2.0;
                    }
                } else if (o.x < x.x) {

                    if (o.y > x.y) {
                        alpha = mathe.atan((double)(o.y-x.y) / (double)(x.x-o.x));
                    } else { // o.y <= x.y
                        alpha = mathe.atan((double)(x.y-o.y) / (double)(x.x-o.x));
                        alpha = 2 * Math.PI - alpha;
                    }
                } else { // o.x > x.x
                    if (o.y > x.y) {
                        alpha = mathe.atan((double)(o.y-x.y) / (double)(o.x-x.x));
                        alpha = Math.PI - alpha;
                    } else { // o.y <= x.y
                        alpha = mathe.atan((double)(x.y-o.y) / (double)(o.x-x.x));
                        alpha = Math.PI + alpha;
                    }
                }
                alpha *= 180.0 / Math.PI; // convert radiants to degrees
                return (int)alpha;
            }
        }
```