RELIABILITY AND TESTING IN VISION-BASED INTERACTION

by

AUDREY J. W. MBOGHO

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2006

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy.

Lori L. Scarlatos

Date — Chair of Examining Committee

Theodore Brown

Date — Executive Officer

Professor Simon Parsons

Professor Ioannis Stamos

Professor Neng-Fa Zhou

Professor Zhigang Zhu

Professor Francine Federman

Supervision Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

RELIABILITY AND TESTING IN VISION-BASED INTERACTION

by

Audrey J. W. Mbogho

Adviser: Professor Lori L. Scarlatos

While reliability is of crucial importance in software engineering in general, it is especially so in human computer interaction (HCI), where the success or failure of a product hinges upon the level of confidence users have in it. This confidence can be increased by a product's demonstrated reliability. On the other hand, it can be quickly corroded when the product behaves in unexpected ways. Software testing is the most effective approach for revealing weaknesses in software so that they can be eliminated and the software made more reliable. But software is rapidly changing, with current trends in human computer interaction leaning towards perceptual interfaces, which accept real-time, complex, and uncertain input data, such as those from cameras, microphones, and tablets. In particular, on-going research activities in computer vision and HCI seem to indicate that camera-based inputs are poised to play an important role in human computer interaction. Furthermore, advances in camera technology and the appearance of inexpensive, consumer-grade products that integrate cameras into computing devices are strong indicators that vision-based interaction is imminent. There is a question as to whether established practices in software testing, where the test data are predictable and have known outputs, are ready for these trends. How can a vision-based system, given

the uncertainty of its input data and the variability of its outputs, be tested so that it produces a result that demonstrates reliability and engenders user confidence?

This thesis highlights the challenges computer vision brings to the testing of software based on it. By looking at it as a parameter optimization problem, this work examines how approaches in traditional software testing can be augmented to adequately address uncertainty in building reliable vision-based interaction. A solution strategy which incorporates genetic generate-and-test techniques is proposed. The viability of this approach is demonstrated with experiments in which objects that are part of a tangible user interface for educational applications are augmented with visual tags. These visual tags are captured by camera and identified through image analysis. This research shows that findings in this domain can be applied in other domains.

ACKNOWLEDGEMENTS

This thesis is not the product of my work alone. I am most grateful for all the help I have received along the way. First, I would like to thank my adviser, Professor Lori Scarlatos, for her help, guidance, and support in conducting the research and in writing about it in this thesis and elsewhere. I would also like to thank my committee for their support, without which it would not have been possible to complete this process. I am grateful as well for those who, for one reason or another, worked with me only briefly, but whose ideas have found their way into this thesis.

Many thanks go to Dr. Ted Brown for bringing Dr. Scarlatos and me together, for his invaluable advice, and for securing for me some much needed financial aid from time to time. I'm grateful also to Dr. Stanley Habib, who similarly supported me in the early years. I'm deeply indebted to Joseph Driscoll, who facilitated all this, and whose wit, guidance, and help, often beyond the call of duty, made going through the administrative maze possible, and even fun.

What I owe my family is impossible to put in words. I am thankful to my parents for teaching me integrity, by word and example, a quality to be cherished at all costs. I hope that I have demonstrated it well in this work. I hope that they would have been proud. I am grateful for their progeny, my brothers and sisters, nieces and nephews, and other relatives, who have loved me intensely and have come to my rescue in practical ways time and again.

Finally, and most of all, I am grateful to God, the source of all good things.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1.

INTRODUCTION

1.1 Background

When we look at the history of computing, one trend that stands out is the steady rise in the importance of the user interface [53]. This can be attributed to two main factors. One is technology. The increased availability of advanced technologies and the falling costs of these technologies have made it possible to support highly complex computational tasks that were previously intractable. The capability is here now, and rapidly growing, to handle computations that require high processor speeds and large amounts of memory. In order to access these ever more complex and more numerous functions within computers, the means of access, the user interface, cannot remain rudimentary—it too has to grow more sophisticated. The second factor that has increased the importance of the user interface is the quest for accessibility. Giving access to all segments of society is one of the main goals of human computer interaction research. The command line user interface of the early days did not give access to all, and it did not need to, since computers were available to only a few select professionals trained to know and remember cryptic commands. Following the advent of the personal computer in the 1980's, computers have become commonplace. Since then, most environments of human endeavor, such as work, school, and even the home, have become ones that require computer use. This has made it necessary for researchers to think about ways to make the user interface easier for everyone. Thus today we have the enormously successful graphical user interface (GUI) with its familiar desktop metaphor and a mouse

for direct manipulation of widgets and content. Yet the GUI is limited in that it is two-dimensional (2D), which makes it unwieldy to use in the manipulation of the real, three-dimensional world (3D). Furthermore, the mouse and keyboard system is designed for a single user, which makes it difficult to use in tasks that require physical collaboration.

Not only are computing systems growing ever more complex, they are also becoming ubiquitous. Ubiquity puts computing in places where traditional modes for interacting with applications are difficult at best and impossible at worst. For example, text messaging using cellular phones is now a widely used means of cheap communication throughout the world. In the developing world, text messaging is often the only affordable means of long-distance communication. Yet typing in a message on the tiny keypad can be quite uncomfortable and time-consuming, which leads users to invent abbreviations that are sometimes misunderstood. Similarly, receiving pictures, videos, and web pages on cellular phones is possible, but the tiny screen is extremely limited in the quality of the images it can display and the amount of information that people can view comfortably. Navigation on these small devices is difficult. It is a lot easier to provide spoken or camera-based inputs to cellular phones, personal digital assistants (PDAs), and other miniature computing devices, than to type into them. But even with regular sized computers, spoken or visual input offers many benefits over keyboard and mouse input. For example, it would alleviate problems such as repetitive strain injury, a very common debilitating hand ailment for which typing is a major risk factor [10, 52].

Another challenge for the user interface is the quest for universality. This is the need to give all populations, including the handicapped, the very young, the very old, and

those in disadvantaged communities, access to technology that is equal to that available to everyone else. Today's typical user interface is not equitable in this sense. Aging, for example, is associated with declining vision and motor skills. This makes it hard for older people to identify small icons on the screen and to target them with the mouse [14]. Young children also face similar challenges. Use of the mouse is difficult for them because their hands are small and their fine motor skills are not fully developed [9].

To address these concerns, technology and research are once again seeking new ways to enhance the human-centeredness of the user interface. The ultimate goal of these efforts is to produce perceptual interfaces [54, 76], which are user interfaces that simulate the ability of humans to sense their surroundings. Instead of the computer sitting idly waiting for users to issue commands, as is the case with the keyboard and mouse, the perceptive computer can, on its own, take in sounds, vision, and other natural inputs, and make decisions accordingly. Such futuristic systems will not be possible until enormous technological challenges have been overcome. In the meantime, however, there are simpler, more modest versions of such systems that can be achieved through narrowing down the problem scope. That is, given the current state of technology, we should seek, not a general solution, but multiple more problem-specific solutions, offering generality as much as possible without sacrificing robustness.

The focus of this work is on vision-based interaction. Specifically, it reports on experiments with tangible user interfaces for collaborative learning environments, where computer vision facilitates the connection between the tangible and the digital worlds.

HCI research shows that tangible user interfaces (TUIs), which blur the distinctions between data and its representations, hold great promise for easier and more

natural interaction [26, 29]. With these interfaces, users input data by manipulating physical objects in a very natural way, drawing on abilities that have been developed since childhood. With traditional computing there is a conceptual gap between the objects with which we interact (keyboard and mouse) and the actions they induce. For example, nothing in his or her prior experience should lead a novice user to think that double-clicking on an icon will cause a document to open. The desktop metaphor and how to manipulate it to achieve desired outcomes must be learned from scratch.

Another issue that TUIs address is the fact that we operate separately in the physical and digital worlds. Each offers certain advantages that the other lacks. But it is often awkward and time-consuming when we have to transition between the two. TUI research seeks to narrow the gap between the two so that users can operate in both environments seamlessly.

Tangible user interfaces can be implemented in a number of different ways. The key thing is to enable the computer to sense physical objects. This can be done through radio frequency identification (RFID), infrared (IR) sensing, echolocation, computer vision, and other ways. Computer vision has the potential for greater impact because digital imaging technology is becoming very cheap and readily available; the other approaches require specialized and, in some cases, expensive equipment. An important advantage that cameras have over other sensors is the ability to track multiple objects simultaneously, limited only by the image resolution. In addition, there exists a wealth of computer vision algorithms, tweaked over decades of research, which should be put to use where possible to bring about better human computer interaction.

1.2 Vision and the Uncertainty Challenge

Even with the current advanced state of technology the pursuit for human-like intelligence in the interface is a lofty goal indeed. Some experimental sensor-based interfaces include cumbersome headgear, gloves, or other contraptions that must be worn and manipulated in specific ways in order for users to communicate their intentions to the computer [53]. Interfaces based on computer vision are attractive in that the sensor (camera) can receive input passively, unobtrusively and without discomfort to the user. Because of falling prices, cameras are now being integrated into devices such as cellular phones and personal digital assistants (PDAs). In order to make vision-based interfaces a success, the main problem that has to be overcome is the uncertainty inherent in image data. Camera technology along with changing environmental conditions such as lighting, temperature, pressure, and humidity cause the same scene to look different each time it is captured as a digital image.

An image identification algorithm typically makes certain assumptions about its inputs. As a result of the uncertainty in the data, however, images may fail to conform to algorithmic assumptions, causing the algorithm to give results that differ from those that were expected. For example, the way that the lighting is distributed can cause parts of an object of uniform color to appear to have several colors. This would result in erroneous recognition results when the recognition is based on color information. Errors in a vision system that assists a pilot or a surgeon can result in loss of life. Errors in a vision system assisting children in a classroom can cause the children to become frustrated and impede rather that facilitate learning. This research seeks to benefit the latter scenario,

investigating how to develop reliable vision-enabled user interfaces for educational settings.

Bearing in mind that a given vision-enabled interface (VEI) cannot deliver the same degree of reliability all the time, we need to be able to determine how a system will perform under a given set of circumstances. In addition, within the ideal circumstances where a given VEI might perform well, we need to determine the set of input parameters that will make this desirable performance a reality. In other words, we recognize two situations that can make an otherwise good VEI perform poorly. One is when it is put to use in a problem domain for which it is not suited. The other is when input parameters are poorly chosen. The main goal of this research is to address the second issue through experimentation with visual tags.

1.3 Motivation

This research was motivated by a practical need that arose while testing a vision-based software library, code-named Anna. This library, described in detail in Chapter 5, was to be used to build educational applications for young children. Access to these applications would be via a vision-based tangible user interface. Anna would capture live, controlled scenes and perform image analysis on behalf of these applications.

While developing Anna, each function was tested informally as it was completed. Such preliminary testing is important in software development as it provides the opportunity to find the more obvious flaws, and it gives the developer an idea about how to proceed. Normally, this type of testing involves simply picking a handful of random input values and seeing whether running the function under test on them gives the results the developer expects. During the development of Anna, it became clear that this

approach to informal testing, which works for regular software, would not work for software that incorporates the use of live images. The reason for this is that the vision-based software requires some inputs of uncertain value, typically representing tolerances and thresholds. Picking random tolerances, more often than not, gave incorrect outputs even when the function was coded correctly (one can inspect the simpler functions and easily tell if they are correct). This frustrating experience made it necessary to seek out an efficient and automatic way to produce suitable tolerances. A genetic algorithm (GA) is presented in Chapter 6 which fulfils this need. Turning to genetic algorithms for help is not a stretch, as there exists a subtle kinship between genetic algorithms and software testing. In the former, we test to learn something about the input, and in the latter we test to learn something about the program. Together, they provide a complete solution when testing software with uncertain inputs.

1.4 Contributions

The contribution of this work is as follows. It tackles the question of how to produce reliable behavior in interactive vision-based systems that take in inherently uncertain inputs. The uncertainty in the image data cited previously is a major obstacle in the development of computer vision-based interfaces. Guaranteeing reliable program behavior while the inputs cannot be relied upon to take any specific values is desirable but extremely challenging. Images from which the same meaning is to be deduced will vary both spatially and temporally.

The scope of the problem has been explored and, through experimentation, it is shown that by constraining the environment in which vision-based interactive systems operate, it is possible to make them reliable for specific applications. The results indicate

that it is necessary to devise problem-specific vision-based solutions in order to maximize their reliability. Further, it emerges that controlled settings are particularly suited to the use of visual inputs. Finally, a framework is devised, based on genetic algorithms, for testing a class of vision-based interfaces, namely, visual tags, for the purpose of ensuring dependable functionality, and this framework is demonstrated to be adaptable to other problem domains.

1.5 Organization

The rest of the thesis is organized as follows. Chapter 2 describes the difficulties encountered in software testing because of the prohibitively large number of possible input combinations. These difficulties are aggravated by the uncertain conditions in which a vision-based system operates. When a vision-based system fails, it is hard to tell whether the assumption and guesses made were incorrect or a bug in the logic of the program caused the failure. In order to understand the vision problem well enough to deal with this problem, it is necessary to constrain it.

Chapter 3 presents visual tags as one way to restrict the vision problem. Various visual tagging schemes used by researchers are presented. Chapter 4 describes the implementations of visual tags devised for this research and suggests a number of uses for them in educational applications. Chapter 5 delves deeper into the specifics of these implementations, discussing the architecture of the tag reading software in greater detail.

Chapter 6 is devoted to the question of how to discover suitable input parameters in order to increase tag reading accuracy in a given environment. The assumption here is that the software is bug-free, and any errors are the result of bad parameter values. Note that parameter values that are good in one environment can be bad in another. A genetic

algorithm (GA) approach for approximating suitable parameter values is presented. The GA is tweaked in various ways by modifying the fitness function. The results of using the GA approach are compared to those from the maximum distance (MD) approach.

Chapter 7 concludes that evaluation through rigorous experimentation and testing is a suitable approach for discovering techniques that work well and the conditions that maximize the chances for optimal performance. Related questions that remain to be explored are posed and suggestions are made regarding directions for future work.

CHAPTER 2.

THE TESTING CHALLENGE

2.1 Introduction

According to a 2002 report from the National Institute of Standards [47], because of inadequate testing, the cost resulting from faults in deployed software is between $22.2 billion and $59.5 billion annually in the United States alone. This estimate does not include special cases such as the catastrophic failures of costly space missions that occur from time to time, some due to software glitches. Even more critical is the loss of human life that can result from faults in software embedded in devices that control vehicles, aircraft, and medical equipment. Spurred on by competition, technological advances have led to the manufacture of powerful, low cost, miniaturized computing devices that make fast processing of real-time signals readily available, a feat that in the past was only achievable in a few select research labs housing supercomputers. As a result, computing has become more pervasive and the software that interfaces with it more complex. However, research into ways to ensure reliability in these more complex software systems through testing lags behind. The inadequacy in current testing practices cited in the NIST report attests to the difficulties in achieving satisfactory testing.

The main difficulty in software testing is that the number of input possibilities renders it impossible to test the software exhaustively. For example, a 10-character input string, where each character is 8 bits long, has over 2^{80} possible values [6]. If one test run took one microsecond, complete testing would take over 38 billion years. Research efforts in software testing focus on the question of how to test sufficiently, given that we can only exercise a small subset of all the possible behaviors of a program. Testing

sufficiently means testing to ensure that all the likely scenarios of usage of the software are free of faults, or if faults are present, the testing reveals them. Deciding what to include in the set of likely scenarios is difficult. It involves anticipating users' intents, some of which might be malicious. Buffer overflow attacks [13] are an example of malicious use of software which was not anticipated. It is also necessary to anticipate inputs that occur in the form of system configuration. For example, a program might make a system call to obtain the current time; proper testing of such programs would have revealed the flaws that led to the Y2K sensation.

Most programs are not replete with errors. Rather they contain a few extremely subtle bugs that escape the sort of informal testing that programmers routinely conduct as they develop. Formal testing is well established as essential in software engineering. The three levels recognized in traditional software testing, namely, unit, integration, and system testing are part of the popular Waterfall Model of the software development life cycle [30]. Corporations and governments devote significant investment to software testing, and academic research in this area remains active year after year. A search on the Web for information on "software testing" reveals that the software industry continues to produce a large number of test automation tools along with publications on the use of these tools or on testing in general [70].

2.2 Testing Levels: When to test

There are three levels of testing in the Waterfall Model, and a fourth appears in [30]. These are unit, integration, system, and interaction testing. While these terms are self-explanatory, a word about integration and interaction testing is in order. Today's systems are highly integrated. Object oriented programming, components, plug-ins, and

such, define the current software development landscape. It often happens that a piece of software which works perfectly on its own fails when integrated with other units. The importance of integration testing cannot be overstated. Similarly, interaction testing is extremely important in today's multi-tasking environments. Failures are known to occur due to conflicts between processes running concurrently. Finally, testing of the finished product must be done with users to assess its usability. Usability testing, while extremely important, is not directly related to software reliability and will, therefore, not be discussed further in this research.

2.3 Testing Methods: How to test

The research defines two main categories of software testing methods: functional testing and structural testing. These are also known as black box and white box testing respectively, referring to the fact that the internals of the object being tested are unknown in the former approach and fully known in the latter. A third category that has emerged is gray box testing, which is a mix of the other two forced by today's software architectures that consist of both known and unknown elements.

2.4 Functional Testing

In functional testing, the software under test (SUT) is viewed as a function that maps inputs to outputs. No knowledge of the internal structure of this function is assumed. The choice of inputs is guided by various criteria that lead to finer-grained categorization, including, boundary value testing, equivalence class testing, and decision table-based testing.

In boundary value testing, the input values chosen for each variable are the minimum, one more than the minimum, the maximum, and one less than the maximum. These values are often denoted min, min+, max, max-, and the rationale in picking them is that programs are often buggy in their handling of domain endpoints. (An extension of this scheme includes min- and max+.) In worst case boundary value testing, which assumes multiple faults, a test case is generated by forming a Cartesian product from the sets of boundary values generated for each variable. This generates 4^n cases (6^n in the extended scheme), where n is the number of variables.

Equivalence class testing is a type of functional testing that groups together input values that have some relevant property in common. Once a member of a given equivalence class has been tested, that class is considered covered. In the triangle problem [30], for example, the program takes three integers that are the lengths of the sides of a triangle. It then determines whether it is equilateral, isosceles, scalene, or not a triangle. All triples that would yield the same answer can be considered to belong to the same class. So it is sufficient to test the triple (5, 5, 5) only, and not also (6, 6, 6), (80, 80, 80), and so on, as all these belong to the "equilateral" class.

Decision table-based testing generates test cases by listing in a table all the conditionals, the combination of all their values and for each combination, what action to take. This technique is suitable for testing programs in which a conditional structure is dominant and if there are dependencies among input variables. On the other hand, if variables are independent or refer to physical quantities, equivalence class testing is more suitable.

A shortcoming of functional testing is that it allows gaps and redundancies. Important values can be skipped, while other values are tested repeatedly. Structural testing, which we look at next, does not suffer from this problem.

2.5 Structural Testing

In structural testing, the tester has full access to the implementation and makes use of it in deciding what test cases to generate. This allows the avoidance of the problem of gaps and redundancies. A second advantage of structural testing is that it makes possible the formulation of metrics that can be used to evaluate structural testing techniques. Two sub-categories of structural testing are path testing and dataflow testing. In path testing, a program graph is constructed, with code fragments represented by nodes and flow of control by edges. Dataflow testing checks whether anything goes wrong from the point where a variable is defined to the point where it is used.

Testing literature defines several different sets of metrics. The following set of five [57] is commonly used.

C_0: statement coverage – the ratio of the number of statements tested to the total number of statements.

C_1: branch coverage – the ratio of the number of branches tested to the total number of branches.

C_2: condition coverage – the ratio of the number of evaluated expressions in conditions to the total number of expressions in conditions

C_3: combinations of conditions coverage – ratio of the number of combinations of conditions tested to the total number of condition combinations.

C_4: path coverage – ratio of tested paths to the total number of paths.

The goal is to maximize one or more of these ratios within the amount of time and computational resources available to a project. This goal is not easy to achieve because when a test case is generated, it is not possible to tell in advance how well it will perform. The metrics can only be reported (through code instrumentation) after the testing is done. Empirical evidence can be gathered in this way and then used to guide future decisions about what methods generate test cases that yield the best coverage.

2.6 Testing in the Face of Continual Change

As new programming paradigms, such as object oriented programming (OOP), have come into widespread use and replaced old ones, and as graphical user interfaces (GUI) have replaced character based interfaces, software testing models have had to be reviewed and changed. An important consequence of OOP is that software is no longer written from scratch. Large portions of it consist of components written by others that are then plugged into a new project. Similarly, the Wide World Web, which allows users to interact with many diverse systems located remotely, has forced researchers to think of software testing in new ways. All these changes, to name but a few, mean that one size no longer fits all in testing; rather, a different approach is required for each new paradigm.

In [19], Doong and Frankl introduce a method for testing object-oriented software using the message exchange metaphor often applied to OOP. Messages are sent to objects of the class under test and the state of each object is observed after processing the messages. A test case consists of two sequences of messages and a tag that indicates whether processing of either sequence should result in the object entering the same state. After receiving the sequences (i.e., executing the test), the actual resulting states are

compared and the result of this comparison is checked against the tag. If there is a match, then the object has behaved as expected. Otherwise, an error in the class has been found.

In [46], Nguyen *et al.* address the specific issues that affect testing Web applications, such as the vast diversity in both hardware and software that the web incorporates. In what is termed a "software mix," a web application may include multiple operating systems, software packages, components, multiple server types, and multiple browser types. In these circumstances testing difficulties are greatly multiplied as an error could result from a great many combinations of factors.

Today, yet another major change in computing is emerging, namely, perceptual interfaces. Perceptual interfaces are ones that react to people's interactions that are presented in spoken, visual, or other perceptible forms. A key difference between perceived inputs and keyboard/mouse inputs is that the degree of uncertainty in the former is far greater, as is the accompanying risk of failure. These new challenges call for software testing models and practices to be reviewed once again. This is by no means a suggestion that old testing methods should be discarded. The developments in software testing that have come about in response to the evolution of computation should be completely relevant to perceptual technologies.

2.7 Test Data Generation

Test data generation continues to be an area of active research. Random data generation is known to be unsuitable for exhaustive testing, as it tends to be costly and yet produces data lacking the required statistical properties [73].

Some recent research views test-data generation as an optimization or a search problem [48, 73, 74]. For example, one might be interested in test-data that cause a

specific statement to be executed, a particular branch to be taken or exceptions to be raised.

In vision-based systems, test data generation is extremely important. The problem of uncertainty, which is our main concern in this thesis, arises directly from the type of data that computer vision must deal with, namely images. In the later chapters, we look at vision based interfaces and how they are tested. But first, we note that the problem of uncertainty affects that entire crop of new interaction modalities, mentioned earlier, collectively known as perceptual interfaces.

2.8 Perceptual Interfaces

Perceptual interfaces present a new challenge for software testing. Perceptual interfaces are ones that can receive and process real-time, natural inputs and provide appropriate responses. Natural inputs can include images from cameras, sound from microphones, and others. Perceptual computing devices are have the potential to become more ubiquitous than keyboard and mouse systems are today as they will include cell phones, PDAs, chips embedded in household appliances, just to mention a few. Shrinking devices are making it harder to input information using keystrokes. It is easier to speak to them, scribble on them, or show them. Speech recognition and pen/handwriting recognition technologies are already in use and fairly well developed even in small devices. Non-trivial processing of real-time images, on the other hand, lags behind. Research in the testing of such systems lags even farther behind [8, 75], and this could be the reason why at present there is little evidence of vision-based interfaces in use outside of research labs. In the next section we discuss the current research that deals with the important question of ensuring reliability in computer vision through testing.

2.9 Testing in Computer Vision

It appears that not enough is being done in this important area [8]. Trucco and Verri give two reasons why testing is important to computer vision [75]. The first reason is that computer vision is experimental because of its inherent non-determinism. To get good results, researchers must experiment with real data and different parameter values. The second is that there is a need to combat tendencies in the field to publish work that is not backed up by sufficient experimental evidence. Trucco and Verri as well as Bowyer give general guidelines to follow in designing a testing and evaluation protocol. First, we need to determine what the key inputs to the algorithm are, and whether it is a measurement algorithm (e.g. one that gives the size of an object) or a detection algorithm (e.g. one that finds out whether an object is present in the scene). We must also predict the expected outcome of the algorithm using ground truth and real data. For measurement algorithms, error estimation formulas from Statistics should be used. For detection algorithms, the numbers of false positives and false negatives should be observed. The error quantities obtained give a measure of the algorithm's correctness and reliability.

Similarly, Bowyer and Phillips [8] make a strong case for the need to develop evaluation and testing methods for vision algorithms. They decry the paltriness of work in this area, especially in contrast to the great need that exists for it given the large quantity of computer vision algorithms and those in related areas such as image processing and image analysis produced over the decades. They suggest that a major reason for this situation is inattention to this issue from journal reviewers and funding agencies. Decision makers in research funding and publishing should insist that results

be backed up by solid experimental evidence obtained through standardized approaches that the vision community agrees on.

Liu et al discuss one of the early efforts applying statistical methods in the validation of vision algorithms [34]. An important distinguishing feature of this work is that it considers coding errors, while other researchers have seemed to focus only on the evaluation of the algorithms themselves. Implementation details cannot be ignored as they can sometimes be the source of incorrect output. Algorithm validation must be augmented with testing of actual implementations. The need for testing in the context of a real application is recognized in [20], where it is listed among the four essential features of an evaluation protocol.

In [63], a large number of dense stereo correspondence algorithms is surveyed and evaluated, the motivation, once again, being the lack of such work in the field. In [31], Kamberova and Bajcsy present a method for modeling intrinsic noise in the process of image acquisition for stereo reconstruction. Their model can be used to measure the degree to which a noise reduction method improves accuracy and precision.

In [20], Dougherty and Bowyer identify four categories of evaluation for edge detectors. These are human evaluation, theory-based evaluation, edge feature measurement evaluation, and ground-truth comparative evaluation. Theory-based and feature measurement evaluation techniques are found to produce unreliable results. While human evaluators can easily see edges that a detector might miss, they can at the same time miss small differences in results that a computer could use to rank two detectors. For example, a human may not be able to tell when the edge image from one

detector is only a few pixels longer than that returned by a poorer detector. Therefore fine-grained sampling of input parameters is of no value to human evaluation.

The technique in [20] is in the ground-truth comparative evaluation category which they find does not have any of these shortcomings, but rather offers important advantages. The use of ground truth permits testing with the kinds of images that the detector is likely to encounter in real situations. Hence this category of methods results in more accurate ranking of detectors.

The technique in [20] samples the parameter space in a systematic, repeatable way. In what they call adaptive sampling, the contribution of a parameter to the computation is taken into account; more important parameters are sampled more frequently. Also, each parameter is varied until no gains in performance can be observed. To arrive at a ranking for a given detector, the edges found by the detector are compared to the ground truth. The best detector is the one with the lowest percentage of missing edges that are present in the ground truth (false negatives) and the lowest percentage of edges found that are not present in the ground truth (false positives). An additional rating is provided by keeping track of the number of times parameters are changed. Not having to resample a detector's parameters too often before reaching peak performance improves the detector's ranking.

2.10 Two-Phase Testing

Testing in computer vision needs to be done with two goals in mind, and therefore, in two phases. One is the usual goal of testing in order to ensure the program is free of errors. This will normally be done in the development environment, where perfect synthetic images should be used to eliminate them as a potential source of failure. The

second goal is what makes testing in vision different from testing ordinary deterministic software. Vision algorithms require many different parameters besides the images themselves. One category of these parameters provides bounds for ranges of values. This identifies the input parameters that are suitable in a given environment. Computer vision software that is bug-free can still fail when placed in an environment different from the one in which it was initially debugged, because of parameters that need to be adjusted and fine-tuned for each environment. Since each environment determines its own parameter values, they are not known beforehand. Thus the second testing goal requires a second, subsequent Phase of testing that is performed in the environment of use. This phase must be repeated whenever the environment changes—for example a vision system is moved from a room in the basement to a well lit higher floor. The task of testing for the right parameter values can be combinatorially prohibitive. Chapter 4 describes how this problem has manifested itself in our experiments with visual tags. Chapter 6 models this as an optimization problem and presents a genetic algorithm for dealing with it.

CHAPTER 3.

VISUAL TAGS

3.1 Introduction

In vision-based human computer interaction, cameras provide the input. Other input methods require users to wear special gloves, put on helmets, or have electrodes attached to their faces [27, 38, 71]. Cameras, on the other hand, are beneficial in that they operate passively, causing no discomfort to the user. In fact, the user need not be aware of the camera at all. However, the information that is input is in the form of images and the computer must interpret these images and draw conclusions about properties of objects in the scene—their appearance, location, orientation, motion, and so on. These tasks are computationally intensive and demand high processor speeds and large amounts of memory. Even with today's 3 GHz processors, several orders of magnitude faster than the supercomputer of yesteryear, the general vision problem remains intractable. A major concern in gesture recognition, for example, is that the image processing and analysis may take up all of the available computational resources and leave none for the actual application. Interfaces that have been built to respond to hand gestures as mouse clicks [51] are not able to replace the mouse since they inevitably are slower because of the processing that is needed to guard against recognition errors. Ye et al.'s [79] approach to this problem in their gesture recognition system is to avoid tracking. Instead, they process gestures only when they are close to the objects that are expected to be handled. Betke et al., in EyeKeys [37], use a separate computer for the vision tasks and then communicate results back to the computer running the main

application. In this case, communication between the two computers can cause unacceptable delays.

In practice this means a trade-off between sophisticated, meticulous techniques that can maximize accuracy, and simpler, more error-prone, but faster ones that can give immediate responses to users. With interactive systems where fast response times are critical the choice is obvious; speed wins over accuracy. But accuracy does not come exclusively from greater thoroughness and sophistication in image processing and image analysis. It can be obtained in other ways. One way is to constrain the problem so that strong assumptions can be made without excessive risk. If the environment in which a vision system operates can be controlled, we can alleviate the problem of imperfect image data by emphasizing the things that we want to detect. A specific instance of this approach is the use of visual tags to identify those things that we want to track.

Visual tags are labels, often made from paper or tape, which are affixed onto objects in order to identify them. Tags can also be printed onto objects directly. A camera captures the visual tag and passes the image on to vision software for processing, analysis, and recognition. As opposed to "hi-tech" tags that use infrared or radio frequency identification (RFID) [56], visual tags are low-cost and they allow multiple objects to be captured simultaneously. Furthermore, while RFID tags require a special reading device, capturing visual tags only needs a camera. Due to falling prices of imaging technology, cameras are readily available. They are increasingly coming integrated into computing devices such as cellular phones and personal digital assistants (PDAs). In addition, reading visual tags is made easy by the availability of computer vision algorithms—the result of decades of research. The criticism that these tags are

visually obtrusive may be valid in general, but in a classroom for young children, colorful representations of concepts are an asset and are completely normal.

A tag can be a plain marker carrying one bit of information (present or absent), or it can incorporate various shapes and colors encoding a binary string of as many bits as the application requires. By tagging, the problem of handling the large amount of data associated with a large and complex image is reduced to one of finding a small object (tag) which stands out in its environment and whose appearance is known a priori. A good tagging scheme should maximize the amount of information it can accurately hold in a small area.

3.2 Related Work

A number of research projects proposing visual tags for a wide range of applications have been reported in the literature.

In TICLE (Tangible Interfaces for Collaborative Learning Environments), Scarlatos [61] uses a labeling scheme to identify physical puzzle pieces in a vision-based educational application. Each puzzle piece is marked with reflective colored spots arranged in a line. One color marks the center of a piece, while the other colors serve to identify the piece and its orientation. Knowing the orientation allows the spatial relationships among pieces to be determined. A camera is used to capture the puzzle as its construction progresses. When requested, the computer vision software translates the current frame into a symbolic representation. The software then determines if the symbolic representation is correct (i.e. corresponds to the solution or a partial solution). If the players are making good progress, the software offers encouragement; otherwise it provides guidance.

Underkoffler's Illuminating Light [77] is a vision-based TUI for helping optical engineers to design holographic layouts. Optical elements normally used in such designs require extremely careful handling as they are expensive and fragile. Illuminating light uses plastic models of these elements. Each model element is marked with a set of colored dots from which the vision software can determine the identity, location, and orientation of the element. Simulated light rays are generated and displayed in response to how users juxtapose objects.

In their collaborage system [45] Moran *et al.* use tags to uniquely identify items on large wall-mounted boards. A camera captures the board and notes events such as removal or movement of items. This work recognizes, as we do, that uncertainty in image identity presents a serious challenge to the reliability of vision-based systems. They tackle this issue by assigning confidence levels (from 0 to 1) to various aspects of the state of the board. The confidence level for an item is 1 only if it is identified when the camera has zoomed in on it. When the camera zooms back, that confidence level begins to decay. Other events noted can cause the decay to accelerate. For example, if there is activity detected around the board when a glyph goes missing, there is a good chance that it was just occluded by someone passing by. But if there was no activity, then it is more likely that the glyph was moved or switched. The latter scenario speeds up confidence decay. The camera zooms in next on the item whose identity is least certain.

Iannizzotto et al. [28] present a system for assisting people with impaired sight to navigate their surroundings. A head mounted camera takes images of objects and a computer worn on the belt finds and reads barcodes from these images. This permits the

identity, location and pose of objects to be determined and the user to be informed about various things, including where objects are for purposes of obstacle avoidance.

TRIP [16] is a system that uses barcodes and cameras to locate users and objects in living and working environments. One proposed application is the use of a video camera to record locations of tagged books in a library and link a book's information in the electronic catalog database with its exact physical location in the stacks. In another application a software agent associated with each person locates that person and migrates an audio player and MP3 decoder to the computer nearest to him. A third use is in recognizing when a user is near a workstation and then logging that user on automatically.

CyberCode [58] is a tagging scheme for use in augmented reality environments. The tag is a 2D black and white barcode that is captured using low-cost CMOS or CCD cameras such as those that come attached to mobile devices. As with TRIP, both the tag and its location are identified. One use is putting the tags on printed documents to provide a link between the hard copy and its electronic version. In another use, the tag, once captured, brings up menu items on the screen that can then be selected by moving the tagged object around.

Kim and Fellner [32] use white paper markers in hand gesture recognition for a back-projection wall. The marked hand performs manipulations and selections that are captured by cameras. This work demonstrates a common practice in vision-based applications—the use of artifacts (paper markers, in this case) to enhance the environment for the sake of the application. This inevitably makes the application unusable for some. For example, because the recognizer looks for the white paper

thimbles, this system assumes that users are not wearing white clothing or shiny jewelry. This would not be a reasonable assumption for general-purpose vision as people very commonly dress this way.

Stafford-Fraser and Robinson [69] present a whiteboard application that uses the board itself as a tangible user interface. Users write symbols on a designated area of the whiteboard which are captured by a video camera mounted overhead. Handwriting recognition is used to identify the symbols. Prolog is used to define relationships between symbols and the actions they are meant to trigger. This is done in order not to hardcode these relationships, thus allowing users to define the semantics of symbols themselves. Audio feedback notifies the user when commands have been executed. The same system can be used to capture the documents on a desk. For example, a user can attach to a document a Post-It note or cardboard cutout with a letter written on it, which is then captured by the camera attached to the user's computer. The computer might then fax the document, send it to the copier, or email it, depending on what letter appears on the tag. Handwriting recognition poses the greatest challenge for this prototype and takes up most of the computational time.

MERL (Mitsubishi Electric Research Laboratories) [5] has built a prototype that uses visual tags for identification, as logos, and to provide information about the wearer. An identity tag allows tracking of people or objects. A logo tag allows a vending machine, for example, to personalize service to the wearer. An information tag can give information about a handicapped person and cause a computerized system, such as an elevator, to provide a special level of service.

3.4 Camera Phones

Following the rapid increase in the computing power available on the average PC, research into gesture recognition became more prevalent, pushing aside visual tags. With the recent arrival of camera phones on the market, however, there is renewed interest in visual tags. These phones come equipped with a significant amount of memory and an operating system and programs can be written for them in common languages like C++ or Java.

In Semacode [66] a smart phone (which always includes a camera) is used to capture a 2D black and white data matrix tag. This type of tag is available in the public domain. Its standard is specified by ISO/IEC (International Organization for Standardization, International Electrotechnical Commision). In the Semacode implementation the tag can hold more than 100 ASCII characters, which represent a URL associated with the object on which it is placed.

Aoki and Matsushita's Balloon tag [3] is based on infrared technology. These tags are equipped with a chip, battery and LEDs that transmit infrared pulses. An ordinary video camera captures the balloon tag and decodes it. The balloon can contain a portion of the Bluetooth address of the user's wireless device. The balloon tag can transmit this address (BD_ADDR) information to a server along with its distance from the server. The server can then select the closest user and connect to that user's Bluetooth device first.

Rohs [59] describes a user interface based on visual tags known as visual codes. Picture menus that are printed on paper or displayed electronically are captured with a camera phone. Analyzing the picture allows identification of various widgets. The user

can then interact with the menu on his/her cell phone and send a short text message (SMS) to the phone number encoded in the menu ID.

Scott *et al.* [65] introduce a system for a camera-equipped device to quickly obtain the Bluetooth address of another device using visual tags instead of the "slow and awkward" Bluetooth device discovery protocol. Tags have a circular design, similar to that of TRIP [16]. A tag can be a physical paper tag or it can be electronically generated and displayed on a device's screen. A phone wishing to communicate with another device captures the tag associated with the other device and analyzes the image in order to decode from it the Bluetooth address of the other device.

Toye *et al.* [72] present a client-server system that uses the connection method just described. Subsequent to being used to discover device addresses, visual tags are used to select advertised services. For example a vacation poster can have a visual tag printed on it. A user wanting more information can capture the visual tag and decode the server's Bluetooth address contained in it, thus initiating communication with the server. The user can select one of several services provided by this server and encoded in the tag. Subsequently the server can obtain permitted personal information from the smart phone, such as a mailing address or credit card number.

CHAPTER 4.

VISUAL TAGGING STRATEGIES FOR EDUCATIONAL APPLICATIONS

4.1 Introduction

The classroom setting is an ideal test-bed for visual tags. Many educational activities involve the identification of objects. For example, playing physical puzzles involves recognition of the puzzle pieces. Tagging the pieces simplifies the automation of this task. Similarly, abstract concepts can be represented using drawings on physical objects (e.g. wooden cubes) which can then be tagged so that the computer can recognize them as well. The basic idea is to have a computer quickly recognize an object, and perhaps also its position relative to other objects, so that a vision-based application can guide, correct, or encourage a child attempting to also identify that object as part of an educational exercise. For each activity, the tagging scheme chosen must allow the representation of enough values to cover all the different objects used.

A suite of visual tagging strategies for educational activities and a library of functions for detecting those tags (described in Chapter 5) have been developed in this research, some of which has been reported previously [40, 41, 61]. Using these strategies, any ordinary object—such as a puzzle piece, math manipulative, or science specimen—can be made part of a tangible user interface. These tags may be used to identify, locate, orient, and find the attributes of puzzle pieces or manipulatives in an environment.

In image analysis, the use of color is often shunned because of the difficulties it presents. These difficulties arise from the wide range of shades a color can take as illumination changes. That is, the problem of uncertainty is worse in the presence of color. On the other hand, color offers additional evidence for or against an object's identity. Humans use color this way all the time, and it would be helpful if machines that assist humans could have the same ability. Therefore the choice was made to use color in this research in order to attempt to reap the benefits that color does offer over black and white or grayscale analysis. Specifically, the YIQ and HSV color models were used, which allow one to analyze and manipulate luminance and chrominance information separately.

Color analysis is similarly used by Quek et al. [55], but with the RGB model, to segment the hand in a system which replaces the mouse with a pointing finger. The RGB model, however, is known to be unsuitable for image analysis as it tends to segment in a way that does not necessarily correspond to meaningful image attributes [11].

The system proposed by Matovic et al. [39] is similar to ours in its use of color and tangible objects as inputs. The physical objects are color cubes that a user arranges on a semi-transparent surface. A webcam under this surface captures this arrangement, which represents a sketch of an image the user wants to retrieve from a database. This QBE (query by example) approach tries to match the appearance and placement of colors in the sketch with images in the database.

One of the goals of the experiments conducted in our research is to discover colors that stand out best in the environment and that are easily distinguishable from one another. We have experimented with several tag designs, which we describe next, mostly

focusing on the barcode as it best illustrates the problem of imperfect data. Unlike TRIP and CyberCode, our tags have a simple design. This simplicity allows for fast image analysis, which is crucial for true real-time interactive computing. The barcode is slightly more detailed than our other tags.

4.2 Single Bit (SB) Tag

The SB tag is identified by its color and the presence or absence of a black patch, about 1/2 the size of the tag, at the center (Figure 4.1). We cut out small squares of colored paper to create SB tags. As they are, these plain shapes represent 0. In order to represent 1, we stick onto the colored shape a piece of black tape about half the size of the colored shape. Using black tape rather than coloring ensures uniformity of color. For each color, the SB tag represents a 1 if it has a patch and a 0 if the patch is absent. We read the SB tag by computing the proportion of the tag's area that is not covered by the black patch. If this proportion is close to one, we conclude that there is no patch and set the tag value to 0. Otherwise, the tag value is set to 1.

SB tags can be used to represent attributes that have two values, such as, "yes" or "no." For example, a classroom activity might require children to separate seashells according to whether they are hinged or unhinged. SB tags can be used for computer vision-based assistance in this activity.

While the SB tag has a very simple design, it is extremely powerful in that, theoretically, it can represent a string of any length. A string of bits can be constructed using many SB tags. If one tag color is used, bits must be lined up so that the position of each piece relative to the bits on its left and right can be known. In this case, the reading algorithm must take into account the orientation of the line along which the bits are

placed. This can be achieved using a special marker for the beginning of the bit string so that strings that are not captured horizontally can be rotated before reading. An alternative approach is to use multiple colors in order to simulate bit position. For example, a green bit anywhere on the viewing surface may be assigned position one, a red bit position 2, and so on. The amount of information that can be represented this way is only limited by the number of colors that can be properly captured and distinguished from one another within the field of view of the camera. On the other hand, this method avoids the overhead of having to determine bit string orientation and perform rotations. Secondly, having to line up bits reduces the flexibility and transparency of the user interface—a goal of HCI is to minimize the user's awareness of the interface and its requirements. Thirdly, falling prices indicate that camera technology that can distinguish small differences in color will soon be affordable to the average classroom.

Very young children (4-6 years old) can learn to identify objects they play with on a viewing surface by hearing the computer say audibly what they are. The computer does this after taking an image of the viewing surface and reading the string of bits with which the object there is labeled. If there are multiple objects, the display can highlight the object whose identity it is currently announcing. (A projector can be used to highlight the physical itself.) A related exercise is one where the computer assists the children to indicate which of two attributes is true about a given object. In this case, the object is labeled with only one bit. The children can spend some time trying to figure out the problem on their own, and then ask the computer to tell them the answer.

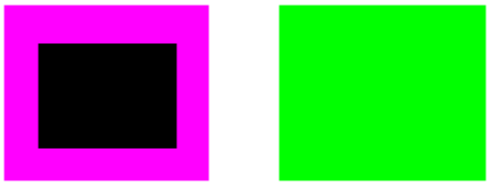

Figure 4.1: One SB tag with value 1 (black patch is present) and one with value 0 (black patch is absent).

4.3 Punch Hole (PH) Tag

In PH tags, shown in Figure 4.2, each tag has a different number of holes punched into it. A tag with no holes represents the value zero. A tag with *n* holes represents the number *n*. A 2" by 1" tag can accommodate up to 15 holes, permitting the representation of 16 values per color. To read PH tags, we construct a lookup table relating pixel counts and the values they represent. We first calculate the area (in pixels) occupied by a tag with no holes. For tags with holes in them, we count the number of pixels of the tag color still remaining on the tag. Thus the largest area represents 0, while the smallest area represents 15, for a given color. Again, the increasing availability of high resolution imaging at low cost will allow PH tags to be made smaller while representing more information. Furthermore, thorough testing, discussed in Chapter 6 will help produce accurate pixel counts.

Figure 4.2: PH tags with values 0 and 3.

While tags can be used in a variety of applications where objects need to be identified, it is useful when the elements of a TUI resemble the things they represent. For example the Illuminating Light TUI [77] uses models that resemble laser guns and mirrors, objects that users of the application recognize easily and readily associate with them the right functions. This argument suggests a use for PH tags in a dice game. Such a game can be used to teach probability, for example.

4.4 Orientation Tags

Orientation tags consist of a line of colored spots (Figure 4.3). One color is used to mark the center of an object, which represents the object's location. The other colors identify the object. The orientation of an object can be determined by calculating the angle between a fixed axis and the line of colored spots. This tagging scheme is used to mark puzzle pieces in the Tangram application [61] (mentioned in Chapter 3), where the orientation information provides a way to draw conclusions about spatial relationships among puzzle pieces. From this spatial information (i.e. where each piece is, how it is oriented, and which pieces are next to it), the application can determine whether or not the puzzle is being solved correctly.

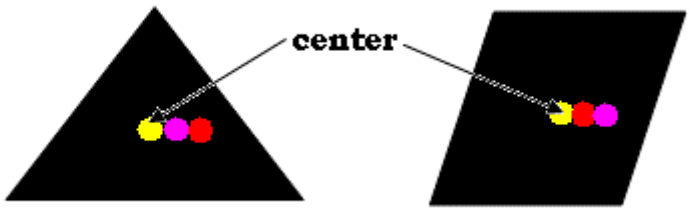

Figure 4.3: Two tagged Tangram pieces. A special color marking the center of each piece gives the location of the piece. A different combination of colors identifies each piece. The linear arrangement of dots helps to determine direction.

4.5 Space Filling (SF) Tags

Certain other puzzles involve filling a given space with the same basic shape, where the basic shape is part of a larger block. Pattern Blocks are shapes that can be constructed from identical basic shapes such as equilateral triangles or squares. Pentomino pieces, for example, are Pattern Blocks made up of different arrangements of the same square shape. A Space Filling (SF) tag is a colored spot marking the center of each basic shape—the square in the case of Pentominoes. Each puzzle piece will have as many spots as the number of its constituent basic shapes. The spots in one puzzle piece are all the same color, thus identifying the piece. The position of each spot gives the location of the basic shape whose center it marks. Figure 4.4 shows an example.

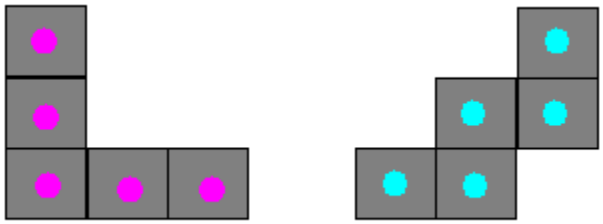

Figure 4.4: Two Pentomino pieces marked at the center with SF tags.

Space filling puzzles present many interesting challenges which can provide learning opportunities. For example, a Pentomino challenge might be to fill a 3 by 20 rectangle. There are two different solutions for this challenge [60]. Using the identity and location information provided by the SF tags, the vision application is able to recognize any of the two solutions, as well as progress, or the lack of it, during construction, and then offer encouragement or suggestions accordingly.

4.6 The Tricolor Barcode

Other barcode-enhanced user interfaces have been proposed, but many are black and white and require a special barcode reading device. For example, in [35], a bookmark management system is introduced for barcoding objects so that they can be linked to the web addresses with which they are associated. Scanning the barcode using a barcode reader causes the current web page address to be stored in a database. Scanning while holding down the Ctrl key brings up the addresses previously linked to the object.

The tricolor barcode consists of three different colored tags, such as, red, green, and blue. On a tag is a barcode made up of black and white bars. The leftmost bar is white. It marks the beginning of the code and also indicates what color represents binary 1. Black bars represent binary zero. The code itself begins at the second bar and is read up to the middle of the tag. The remaining half of the tag is a mirror image of the first half, so that the code can be read in either direction. Figure 4.5 shows an example of a tag representing binary 9.

Figure 4.5: Barcode representing binary 9 (1001)

4.6.1 Sorting

The tricolor barcode can be used in a sorting classroom activity for children in the 6 to 8 age range. Such an application has been described in [41]. Sorting involves asking children to take miscellaneous objects and sort them according to some predefined category. Tags such as the ones shown in Figure 4.6 are affixed to each object. Each tag color represents a category or attribute, such as, rock density, origin, type and so on.

Figure 4.6: Each barcode color represents a specific attribute. The number encoded by the barcode represents the value of that attribute for this object.

The children place the objects on a mounted glass surface with a camera below [62]. Under the glass is a camera that continuously sends a live video stream to the computer running the sorting application. When the application receives a request to process the current image, a sample is grabbed off the stream and analyzed in a manner, described below, that ensures a tag can be read regardless of its orientation.

4.6.2 Barcode Reading Algorithm

Assuming for the sake of clarity that we are looking at red tags in the captured image, we first find the bounding boxes for all red tags in the image. These are encoded in a list, which is then saved to a file. Each 4-tuple in the list represents the minimum row, maximum row, minimum column, and maximum column values for a bounding box. Tags are read by analyzing the colors and color transitions inside the area enclosed by each bounding box. The tags may appear in the image oriented in any direction. The tag reading algorithm, adapted from [49], takes this into account by following the steps below.

1. We find the bounding boxes of barcodes of color *x* (*x* is red in Figure 4.5 above).

2. We scan the bounding box in at most four directions — top to bottom down the middle, left to right across the middle, top left corner to bottom right corner, and bottom left corner to top right corner. We stop with the direction that finds a valid tag (only the correct direction will find all 10 bars). Because the barcode is bi-directional, it is not necessary to scan in the other four directions that run opposite these. This significantly reduces computational overhead.
3. We read and record each bit encountered while scanning. Reading involves counting the color transitions that indicate that a bar (other than the guard bar) has been completely crossed. Thus in figure 4.5, starting from the second white bar, we note the transitions white-to-red (1), black-to-red (0), black-to-red (0), white-to-red (1).
4. We stop recording the bits read as soon as four are found, which means the middle of the tag has been reached. Scanning does continue to the end, however, in order to determine the validity of the tag. If 10 bars are found, the procedure reports the recorded bit string and exits. If a direction finds fewer than 10 bars, the recorded bits are ignored and the procedure begins again in another direction.
5. If fewer than 10 bars (or 20 transitions) are seen after all four directions have been scanned, an error code is returned indicating an invalid tag.

4.7 Dealing with Imperfect Data

Initially, the system does not know what a red, green, or blue tag is. We have to train it to recognize these colors. We do this by taking a few pictures of the tags at the beginning. We click on a tag of each color multiple times. We have found that about 36 to 40 clicks are sufficient. The average YIQ triple is computed. To compute the

luminance tolerance, we find the maximum absolute Y difference between all samples and the average. We compute the chrominance tolerance by finding the maximum square difference between the (I, Q) pairs of the samples and that of the average. We use this "farthest from midpoint" heuristic in order to avoid missing any pixels of a given color, but outliers can skew it. (Chapter 6 presents an in-depth study of this problem, formulating it as a parameter optimization issue that is amenable to genetic search.)

There are many sophisticated segmentation methods in the literature, as surveyed in [67], but those are best suited to complex scenes. Applying them to visual tags that are by and large uniform would be overkill. Instead we use a straightforward implementation of the contour tracing algorithm due to Pavlidis [50] to find tags.

Contour tracing is sensitive to discontinuities that inevitably appear in the image due to the error prone nature of image acquisition cited earlier. As a result, we often find a much larger number of tags of a given color than are actually present. A partial solution to this problem lies in the fact that many of these are clearly invalid tags consisting of one pixel, a short straight line, or a small rectangle. The tag reading procedure will eliminate these, as it will not find the right number of bars in any of them. We consider this a partial solution for two reasons. The image can be of such poor quality that it is split into many small tags, none of which are valid. Secondly, when contour tracing produces a large number of tags, the application slows down noticeably. It is important to overcome this obstacle in our application, as children can easily grow impatient and abandon the system if it is slack to respond. Choosing the right tolerance values is crucial to solving this problem.

Another method that we have used to come up with suitable luminance and chrominance tolerance values is through manual inspection of images. We look at the image of a tag and if we see a wide variation in brightness over the tag, we use a larger value for luminance tolerance. Similarly if we see a wide variation in the color of the tag, we use a larger value for the chrominance tolerance. We have found that this technique results in tags being read correctly for a significant proportion of the trials. However, it is not immediately clear how to automate this process.

4.8 Experiments

We tested the barcode application by reading tags repeatedly. We varied the experiment in four ways: (1) using a web cam with a direct view of the tags, (2) using a web cam viewing tags through Plexiglas, (3) using a digital video camera with direct view, and (4) using a digital video camera through Plexiglas. Ideally, for a single tag, only one bounding box should be found. This was the case when the application was tested with perfect synthetic images. Multiple bounding boxes are found in real images when degraded quality results in color discontinuities. Thus the number of bounding boxes found per tag is a good indicator of accuracy: the smaller this number, the more accurate the system. The proportion of trials in which tags are read correctly provides additional evidence with respect to accuracy.

Table 4.1: SB Tag Identification and Reading Accuracy

	Bounding boxes per tag	**% Correct readings**
Webcam - direct	1	100
DVC - direct	1	100
DVC - Plexiglas	20	50

Table 4.2: PH Tag Identification and Reading Accuracy

	Bounding boxes per tag	**% Correct readings**
Webcam - direct	1	50
Webcam -Plexiglas	1.18	72
DVC - direct	1	62.5

Table 4.3: Tricolor Barcode Identification and Reading Accuracy

	Bounding boxes per tag	**% Correct readings**
Webcam - direct	1.5	90
DVC - Plexiglas	2.36	10

The results, summarized in tables 4.1-4.3 above, show that, while the use of a Plexiglas surface is more convenient, direct view of objects affords greater accuracy. For applications that use schemes such as SB tags, in which pixel values are aggregated, degraded image quality arising from the use of Plexiglas is less significant as an obstacle to accurate readings. However, where a single misrepresented pixel can break the application, for example in reading the color barcode, Plexiglas is not suitable. We have tried to use clear glass and found that it is worse than Plexiglas as it reflects objects from

below and this obscures the objects of interest that are on top. Our current setup, however, does not allow for objects to be viewed directly; they need to rest on top of something transparent. Another possible setup would be an upright structure with shelves on which to place objects from one side so that a camera on the opposite side can see them directly.

A rather surprising observation we have made is that the digital video camera images are of a lower quality, from a computational analysis point of view, than those obtained from a low-cost, low-resolution webcam.

We note also that the results for PH tags are inconclusive, with unacceptably high percentages of incorrect readings in all test cases. The pixel counts should be strictly decreasing as the number of holes increases. This, however, is not always the case in experiments, and it is another strong indication of the need to optimize parameter values.

CHAPTER 5.

ANNA: AN IMAGE ANALYSIS LIBRARY FOR READING VISUAL TAGS

5.1 Introduction

Our TUI encodes object attributes on the visual tags, captures them with a camera, and reads them. The test applications are implemented as Macromedia Director movies with Lingo scripting, while the image analysis module is implemented in C++ as a Lingo Xtra. This chapter describes the architecture of Anna, an image analysis function library developed in this research for reading a variety of visual tags for use in educational applications. The color models used in tag identification procedures are discussed, and the justification for these choices is presented.

Anna is a Macromedia Director Xtra, which is a dynamic link library (DLL) written in C++ that plugs into Director. While C++ is a powerful language, it does not offer support for the kind of rapid prototyping required by user interface development. Director, on the other hand, is an excellent platform for quickly putting together state of the art multimedia presentations for e-learning. The elements of a Director movie can be manipulated dynamically with Lingo, Director's scripting language. Macromedia Open Architecture (MOA) [36] permits DLLs to be written in C++ to extend Director's functionality, including the command set of Lingo.

5.2 MOA

MOA is based on Microsoft's COM (Component Object Model), a software development paradigm that allows diverse objects to communicate with one another.

This, for example, is what enables an image created in Paint to be embedded and manipulated in a Word document. But whereas COM provides general purpose behavior under the Windows family of operating systems, MOA is tailored to the needs of Macromedia products.

The central concept of MOA (and COM) is the interface. The interface is a set of function prototypes or function declarations without definitions. (In COM parlance, an interface that has been implemented is still called an interface, which can be confusing because in object oriented programming an interface is strictly empty.) In an Xtra, one takes the interfaces that declare the desired behavior and creates classes that define this behavior by implementing the methods of the interface. This allows different implementations for different forms of the same concept. An application then creates objects or instances of the Xtra that provide a yet more specific realization of the interface. An application provides interfaces and also defines how Xtras must construct their interfaces in order to interact with it. The two types of interface, those offered by the application and those offered by the Xtra (and defined as prescribed by the application), hook together to realize the interaction between the application and the Xtra.

5.3 Anna Functionality

In a sorting activity, students place objects on a TICLE table [61] and the application determines whether they have been classified correctly. When tagged objects are ready to be identified a user activates a Read widget in the Director movie. In the current implementation this widget is a button that is activated by a mouse click. In fact, the reading process can be triggered a number of ways, including speaking a command, touching a screen, or by an internal timer event. The Read command invokes a Lingo

script that contains calls to functions in the Xtra. The first function invoked is `getBoundingRecs`, which gets the bounding rectangles of all the tags of a given background color and returns a list of 4-tuples specifying minimum row and column, and maximum row and column for each rectangle found.

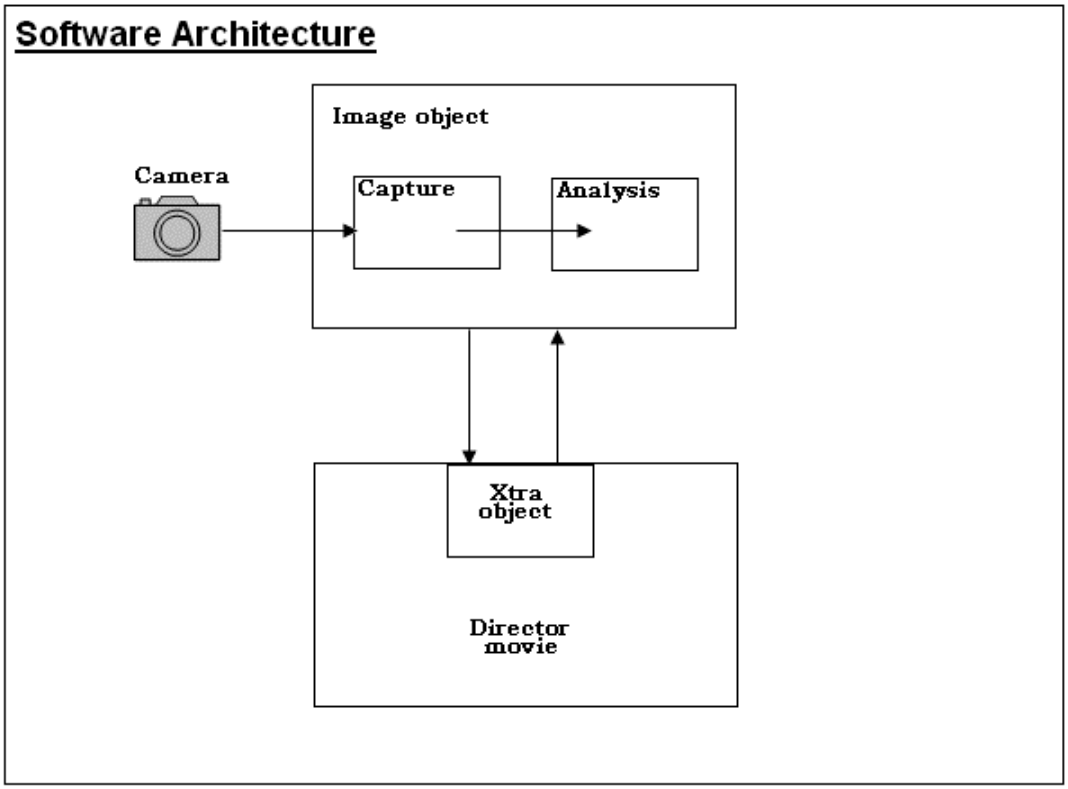

Figure 5.1: An instance of Anna is created as soon as the movie launches. This allows functions of the image analysis library to be invoked at any time. The object is destroyed upon exit from the movie.

The list of bounding rectangles is passed to the specific function for reading the type of tag that is on the current set of tangible objects. This is known to the application since each tag type is associated with a different application. This can be modified so that users select the type of tags they need to read, which would then cause the

appropriate application for those tags to launch. The tag reader processes the bounding rectangles one at a time and returns a list of values.

Anna consists of functions that allow images to be captured and analyzed and tags found in them to be read in real time. While not every frame is processed since images are grabbed upon user request, timing is still a crucial issue because responses must be immediate. Anna's functions therefore use less thorough techniques than ones that could keep users waiting. Accuracy is achieved through careful selection of suitable tolerances and thresholds, a process discussed in greater detail in chapter 6 (Parameter Optimization). The following table is a summary of Anna's functions.

Table 5.1: Functions of Anna

Function	Description
`getImage`	Capture image from camera. Implemented using Microsoft DirectShow video capture.
`getImageFromFile`	Load image from file. Implemented using C++ file I/O.
`showImage`	Create a Director cast member from the captured image (or image loaded from file). MOA cast member manipulation interfaces are used.
`segmentImage`	Segment the image displayed on the Director stage according to an input color. This function is used for testing whether colors have been identified accurately. MOA image manipulation interfaces are used.
`countPixels`	Count the number of pixels of a given color within a given bounding rectangle.
`getBoundingRecs`	Return coordinates of all rectangles bounding regions of a given color in the image.
`getBoundingBox`	Return coordinates of one region of a given color.
`readBarCode`	Return integer encoded in the barcode of a given background color and enclosed in the given bounding rectangle.
`readTag`	Return integer encoded in the barcode. This function is the same as readBarcode except that it analyzes the original Windows bitmap rather than the Director cast member obtained from it.
`readBit`	Return 1 if tag is one solid color; return zero if it contains a patch of another color. The ratio of the number of pixels found to those expected in a whole tag is used to decide whether or not a patch is present.
`readBits`	Return bit values of multiple regions of a given color.

5.4 Color Models

The RGB model, which consists of the three primaries red, green, and blue, corresponds directly to graphics hardware. The transformation of digital color information into a physical display is straightforward. It might seem viable to compare colors in RGB space by a Euclidian similarity measure. Alternatively, one might specify a permissible range for each primary. However, both these approaches result in many visually qualifying colors being left out, and many others being included that do not qualify [11]. While the design of alternatives to the RGB color model, such as YIQ and HSV, is for the most part motivated by human factors (they are intended to enhance the human experience in interacting with computer graphics.), they have proved suitable for machine processing as well.

The color models used in this research are YIQ and HSV. Thus there are two versions of each image analysis function listed above, one for YIQ and one for HSV. An important advantage of these color models over RGB is that they permit brightness and color information to be examined independently of one another.

YIQ is the United States NTSC (National Television Standards Committee) standard for broadcast television and has been in use since 1953 [64]. It was designed to represent color in a way that would be compatible with black and white television. Thus brightness or luminance, the only signal to be received by a black and white TV set, is represented separately in the Y component and color or chrominance is encoded in the I and Q components. A further design goal was to conserve bandwidth by excluding as much as possible color information that the human eye cannot perceive. I encodes a

blue-green to orange vector and Q encodes a yellow-green to magenta vector. The conversion from RGB to YIQ is performed according to the following formula.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & 0.275 & 0.321 \\ 0.212 & 0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

HSV was invented in 1978 by Alvy Ray Smith [68]. HSV attempts to model an artist's way of mixing pigments. He or she picks a pure hue and then lightens it by adding white, darkens it by adding black, or creates an intermediate tone by adding a grey (some mixture of black and white). H stands for hue. S, which stands for saturation, is a measure of the purity of the hue or, in other words, a measure of a hue's departure from gray. Higher saturation values correspond to less gray in the hue. More gray (or less saturation) gives washed out, less pure colors, for example, pastels. Value represents intensity or brightness and measures a hue's departure from black. Darker colors have a lower value. To convert a color representation from RGB to HSV, we let MAX = max(R, G, B) and MIN = min(R, G, B) and perform the following computation:

$$H = \begin{cases} \frac{60(G-B)}{MAX-MIN} + 0 & \text{if } MAX = R \\ \frac{60(B-R)}{MAX-MIN} + 120 & \text{if } MAX = G \\ \frac{60(R-G)}{MAX-MIN} + 240 & \text{if } MAX = B \end{cases}$$

$$S = \frac{MAX - MIN}{MAX}$$

$$V = MAX$$

H is an angle in the range [0..360], while S and V are in the range [0..255] (or [0.0..1.0] when R, G, and B are normalized).

We take advantage of the fact that the YIQ and HSV color models permit the use of visual information in comparing colors. For example, for colors that are highly saturated, one can fix a threshold for S, so that for a given hue, S above a certain threshold is considered a match. This simplifies the comparison and reduces computational complexity. Similarly, with YIQ, the choice of tolerance values can be guided by noticing whether or not, for what should be considered the same color, there are significant changes in luminance or chrominance. If due to lighting conditions, for example, reds appear as both darker reds and pinks, then a larger range of luminance values qualifies. By including image analysis functions for both color models, users of the system can decide, with the help of guidelines, which ones to use for building applications in their environments. While experiments did not include a study comparing YIQ and HSV, informal observations show HSV to be superior where bright and highly saturated colors are used. Experiment 3 in Chapter 6 confirms this observation.

5.5 Anna Usage

This section gives some examples of how Anna is used in a Lingo script. The first step is to create an Anna object, which then makes it possible to invoke its methods. The general algorithm is as follows:

1. Create an instance of Anna.
2. Acquire an image.
3. Get bounding rectangles for all tags of desired color present in the image.
4. Read the tag enclosed in each bounding rectangle.

The first method invoked in all applications is `getImage`. This takes a camera image and stores it in a bitmap in memory. The method `showImage` is then called to create a Director cast member. The original bitmap persists in the Xtra object and it can be analyzed, or the cast member can be analyzed, with identical results. The choice depends on which is most convenient or efficient for a particular task. For example, a cast member can easily be recreated from the original bitmap, so processing that alters the image, such as segmentation, is best performed on the cast member. On the other hand, MOA image manipulation for cast members is quite slow. Therefore, a compute-intensive task such as getting all the bounding rectangles for multiple tags of the same color is best performed on the original bitmap. To summarize, the Lingo code snippet below shows the initialization and clean-up common to all applications.

```
-----------------------------------------------
--Creating an instance of the Anna Xtra
--A standalone movie script
-----------------------------------------------
global tr -- Anna object

on startMovie

  -- create and initialize Xtra object
  set tr = new(xtra "Anna")
```

```
  if objectP(tr) then
    put "Created xtra object successfully"
  else
    put "Error creating object from Anna"
    halt
  end if
end

on stopMovie
  set x = 0 -- Destroy Anna object
end
```

Once the image is captured and a copy of it is in a cast member, the bounding rectangles are obtained with a call to `getBoundingRecs`. The arguments to `getBoundingRecs` include a color and tolerance values, which are read from a color file created previously. This file contains five fields: R, G, B, Y Tolerance, and IQ Tolerance. The computation of tolerance values is addressed in Chapter 6. Following are Lingo sample scripts showing examples of how tags can be read once all the input variables are in place.

5.5.1 Reading Tri-Color Barcodes

```
global tr -- barcode reader object

on mouseUp
  -- Take a picture
  retValue = 0
  retValue = getImage(tr)
  if (retValue = 1) then
    put "Image captured successfully!"
    showImage(tr, 1) -- put image in cast member 1
  else
    put "Image capture failed. Is the camera connected?"
    halt
  end if

  set fname = the moviePath & "barcode.ini"
  put fname --DEBUG

  -- Create an instance of FileIO
  fp = new(xtra "fileio")

  -- Open the file with read only access
  openFile(fp, fname, 1)
```

```
  set ferror = status(fp)
  if ferror then
    alert ("Error opening file" & RETURN & error(fp, ferror))
    halt
  end if

  buffer = readFile(fp)
  thisColor = [0, 0, 0]
  yTol = 0
  iqTol = 0

  -- Loop through each line in file and look for barcodes of that color
  nlines = buffer.line.count
  repeat with i = 1 to nlines
    if (buffer.line[i].word.count = 5) then
      thisColor[1] = integer(buffer.line[i].word[1])
      thisColor[2] = integer(buffer.line[i].word[2])
      thisColor[3] = integer(buffer.line[i].word[3])
      yTol =  integer(buffer.line[i].word[4])
      iqTol = integer(buffer.line[i].word[5])
      bRecList = getBoundingRecs(tr, thisColor, yTol, iqTol)
      put bRecList -- DEBUG

      -- Read barcodes of this color
      nbRecs = bRecList.count / 4
      stride = 4
      bRec = [0, 0, 0, 0]
      repeat with j = 1 to nbRecs
        repeat with k = 1 to 4
          ind = stride * (j - 1) + k
          bRec[k] = bRecList[ind]
        end repeat
        put readBarCode(tr, 1, thisColor, bRec, yTol, iqTol)
      end repeat
    end if
  end repeat

  closeFile(fp)
end
```

5.5.2 Reading SB Tags

```
global tr -- tag reader object

on mouseUp
  -- Take a picture
  retValue = 0
  retValue = getImage(tr)
  if (retValue = 1) then
    put "Image captured successfully!"
    showImage(tr, 1) -- put image in cast member 1
  else
    put "Image capture failed. Is the camera connected?"
    halt
  end if

  set fname = the moviePath & "SBTag.ini"
```

```
  put fname --DEBUG

  -- Create an instance of FileIO
  fp = new(xtra "fileio")

  -- Open the file with read only access
  openFile(fp, fname, 1)
  set ferror = status(fp)
  if ferror then
    alert ("Error opening file" & RETURN & error(fp, ferror))
    halt
  end if

  buffer = readFile(fp)
  thisColor = [0, 0, 0]
  yTol = 0
  iqTol = 0

  -- Loop through each line in and look for and read tags of that color
  nlines = buffer.line.count
  repeat with i = 1 to nlines
    if (buffer.line[i].word.count = 5) then
      thisColor[1] = integer(buffer.line[i].word[1])
      thisColor[2] = integer(buffer.line[i].word[2])
      thisColor[3] = integer(buffer.line[i].word[3])
      yTol =  integer(buffer.line[i].word[4])
      iqTol = integer(buffer.line[i].word[5])
      put readBits(tr, thisColor, yTol, iqTol)
    end if
  end repeat

  closeFile(fp)
end
```

5.5.3 Reading PH Tags

Reading PH tags involves a call to `countPixels` and comparing the returned value with the contents of the previously created PHCounts.ini file. This file, which contains three fields, Value, Low Count, and High Count, is examined line by line. If the returned count falls between Low Count and High Count the, Value at that line is output as the value of the tag and the loop exits.

```
global tr -- tag reader object

on mouseUp
  -- Take a picture
  retValue = 0
  retValue = getImage(tr)
  if (retValue = 1) then
    put "Image captured successfully!"
    showImage(tr, 1) -- put image in cast member 1
  else
    put "Image capture failed. Is the camera connected?"
```

```
  halt
end if

set fColors = the moviePath & "PHColors.ini"
set fCounts = the moviePath & "PHCounts.ini"
put fColors --DEBUG
put fCounts --DEBUG

-- Create an instance of FileIO
fp = new(xtra "fileio")

-- Open the file with read only access
openFile(fp, fColors, 1)
set ferror = status(fp)
if ferror then
  alert ("Error opening file" & RETURN & error(fp, ferror))
  halt
end if

clrBuffer = readFile(fp)
closeFile(fp )
thisColor = [0, 0, 0]
yTol = 0
iqTol = 0

openFile(fp, fCounts, 1)
set ferror = status(fp)
if ferror then
  alert ("Error opening file" & RETURN & error(fp, ferror))
  halt
end if

cntBuffer = readFile(fp)
closeFile(fp)
val = 0
lowCnt = 0
highCnt = 0

-- Loop through each line and look for tags of that color
nlines = clrBuffer.line.count
repeat with i = 1 to nlines
  if (clrBuffer.line[i].word.count = 5) then
    thisColor[1] = integer(clrBuffer.line[i].word[1])
    thisColor[2] = integer(clrBuffer.line[i].word[2])
    thisColor[3] = integer(clrBuffer.line[i].word[3])
    yTol =  integer(clrBuffer.line[i].word[4])
    iqTol = integer(clrBuffer.line[i].word[5])
    bRecList = getBoundingRecs(tr, thisColor, yTol, iqTol)
    put bRecList -- DEBUG

    -- Read barcodes of this color
    nbRecs = bRecList.count / 4
    stride = 4
    bRec = [0, 0, 0, 0]
    repeat with j = 1 to nbRecs
      repeat with k = 1 to 4
        ind = stride * (j - 1) + k
        bRec[k] = bRecList[ind]
      end repeat
      pCount = countPixels(tr, 1, thisColor, bRec, yTol, iqTol)
      mlines = cntBuffer.line.count
```

```
        repeat with p = 1 to mlines
          if (cntBuffer.line[p].word.count = 3) then
            val = integer(cntBuffer.line[p].word[1])
            lowCnt = integer(cntBuffer.line[p].word[2])
            highCnt = integer(cntBuffer.line[p].word[3])
            if (pCount > lowCnt) then
              if (pCount <= highCnt) then
                put val
                exit repeat
              end if
            end if
          end if
        end repeat
      end repeat
    end if
  end repeat
end
```

CHAPTER 6.

INPUT PARAMETER APPROXIMATION

6.1 Introduction

Vision algorithms depend on various parameters of uncertain value. Normally, these parameters are arbitrarily assigned values that are assumed to be suitable. In our experiments with visual tags, these parameters were in the form of Y tolerance, and IQ tolerance for the YIQ color model. For the HSV color model, computations depended on H tolerance, S threshold, and V tolerance. Another metric is used for single bit tags, where the proportion of the tag that is of a certain color determines its value; we need to define the bounds for the range of proportions that constitute a 1 or a zero. In PH tags, a pixel count corresponds to a tag value—a many to one correspondence. A parameter determines the set of pixel counts to be assigned to each pixel. Table 6.1 gives a summary of the parameters used and their meanings.

Table 6.1: Parameters to be Optimized

Parameter	Color Model	Similarity Measure
Y Tolerance, T_y	YIQ	$\text{Abs}(Y_1\text{-}Y_2) \leq T_y$
IQ Tolerance, T_{iq}	YIQ	$(I_1\text{-}I_2)^2(Q_1\text{-}Q_2)^2 \leq T_{iq}$
H Tolerance, T_h	HSV	$\text{Abs}(H_1\text{-}H_2) \leq T_h$
S Threshold, T_s	HSV	$S > T_s$
V Tolerance, T_v	HSV	$\text{Abs}(V_1 - V_2) \leq T_v$

In Phase 2 testing we need to find out which parameter values result in test tags being read correctly in a given environment. The relationship that maps parameters to tag readings is not one that is easy to analyze mathematically in order to minimize deviation from correct readings. On the other hand, testing exhaustively in order to find the best values for these parameters is infeasible in the general case. We consider two approaches to the problem of finding suitable parameters. The first uses what we call the *maximum distance* heuristic, while the second is a genetic generate-and-test approach.

6.2 Maximum Distance

The maximum distance (MD) approach proceeds as follows.

1. Collect sample images in the *environment of use*.
2. For each tag color sample 30 to 50 pixels.
3. Compute the average red, green, and blue values to obtain a representative pixel for the tag color.
4. Compute the YIQ components for the representative pixel.
5. Compute the YIQ components for each pixel in the sample.
6. Compute, according to the appropriate formulas in the table above, the difference between the YIQ values of the representative pixel and the YIQ values of each pixel in the sample to obtain a set of differences $(Y_{av}\text{-}Y_i)$ and $(I_{av}\text{-}I_i)^2(Q_{av}\text{-}Q_i)^2$.
7. Set $T_y = \max(\text{abs}(Y_{av}\text{-}Y_i))$.
8. Set $T_{iq} = \max((I_{av}\text{-}I_i)^2(Q_{av}\text{-}Q_i)^2)$.

This approach works well when there is no great variability in color. Otherwise, it has the potential for just one atypical pixel in the sample to skew the results, which

would then lead to inclusion of unwanted regions and misreading of tags. Experimental results for MD were shown in tables 4.1-4.3 in Chapter 4.

6.3 Genetic Generate-and-Test

Parameters produced through heuristic approaches are risky when the assumptions on which the heuristic is based fail to hold. In MD, for example, the computed tolerances are only guaranteed to find all the pixels that contributed to this computation. The genetic generate and test approach produces parameters that have actually been tested and found to work. This may raise the concern that parameters produced in this way are necessarily tweaked for the sample data, and that they may produce poor results with real data. While this can be an issue in some cases, it is not in the case of the visual tagging scheme set forth here and in the constrained application environments for which it is designed. The following characteristics make it is possible to obtain parameters that ensure a robust visual tagging scheme.

- Distinct colors for tags make over-segmentation unlikely. Therefore tolerances can be made large enough to avoid under-segmentation, which is more likely.
- A stable, homogeneous environment that changes little, particularly in its lighting.
- Generate and test production of parameters in the environment of use.
- The GA implementation is efficient enough for frequent usage if necessary (although, considering the stability of the environment, it is not expected that parameters will need to be re-sampled often).

Genetic algorithms are ideal for tackling optimization problems of exactly this kind, where an analytical solution is not known to exist and where the absolute optimum is not necessary, but a reasonable, “good enough” value is sought instead.

6.4 Overview of Genetic Algorithms

Introduced by John Holland in 1975 [24], genetic algorithms have since gained a great deal of popularity in many disciplines. Many sources of information on the subject exist, for example [21, 44] are good texts. In genetic generate and test, a set of possible solutions is generated and then each is evaluated for possible participation in the production of future candidates. The main difference between this and random generate and test is in how possible solutions are generated, and this seems to make all the difference. The genetic method of generating possible solutions is modeled after the concepts of natural selection and survival of the fittest. Studies have shown that random methods tend to produce poor results (for example [73]). Besides computer scientists, others, such as physicists, geologists, economists, to name a few, have become avid practitioners of genetic algorithms, finding better solutions to optimization problems in their respective fields with genetic algorithms than with traditional heuristic methods or random search. While the theoretical underpinnings of genetic algorithms are hard to capture, it is well-established through empirical evidence that they work.

An important feature of genetic algorithms is that they make little or no use of the specifics of a problem domain, thus making them a black-box testing approach. Therefore, it is possible to attain a very high degree of generalization, so that problems that internally are vastly different can use the same genetic algorithm. Genetic algorithms "blindly" accept as input abstract representations of a problem's possible solutions. The genetic algorithm only knows how fit a representation is; the meaning of it is hidden. This is very useful for the vision problem since there are many factors that are unknown, such as lighting conditions that affect the values that parameters should

take.

Genetic algorithms (GA) belong to the category of algorithms sometimes known as natural algorithms [23]. These are algorithms that mimic nature's optimization techniques. Others in this category include simulated annealing, classifier systems, and neural networks. Such techniques have shown great success in optimization problems that have no known analytical solution. While these approaches may not produce the global optimum, many real-life problems do not require it anyway. For example, an actual traveling salesman may just want a route that will get him to each city on time for a meeting. If a better route exists that gets him there several hours early, it is not required in this case, and so time should not be wasted looking for it. (Of course, the Traveling Salesman Problem is hardly about a traveling salesman at all, but the argument should hold in real applications such as packet routing.)

Studies in genetics have shown that, in successive generations, populations of living things optimize themselves for survival in their environment. Changes introduced into the population by the reproductive processes of crossover and mutation play a key role in this optimization. Genetic algorithms research seeks to model these processes which we now discuss briefly. Much of this information is taken from [7]. A fuller treatment of the subject of genetics is available in any number of biology textbooks.

6.5 Genetics Overview

An organism's characteristics are encoded in its genes, the basic unit of heredity. Each gene contains some molecular permutation of the chemicals adenine, thymine, cytosine, and guanine, which are denoted A, T, C, and G, respectively. The entity on

which genes reside is known as a chromosome. A gene that encodes a given characteristic occurs at a specific location, known as a locus, on the chromosome.

Of interest to GA research are the genetics of higher organisms, that is, organisms, such as humans, that reproduce sexually because this is the type of reproduction that can bring about variation. Variation is essential to the success of genetic algorithms since it is what makes exploration of the solution space possible. On the other hand, with asexual reproduction individuals simply clone themselves so that their population remains homogeneous from one generation to the next, unless they mutate (mutation is discussed later).

The reproductive cells (gametes) have half as many chromosomes as the body cells, and when they combine to form offspring they form a single cell containing the correct number of chromosomes. Growth occurs through cell division. New body cells are formed by a process known as mitosis, in which the chromosomes duplicate and then the cell divides into two, each new cell receiving a copy of each chromosome. Since the gametes must have half the number of chromosomes of the body cells, they cannot be produced through mitosis. Instead, gametes are produced by a process known as meiosis, which is what GA's model.

6.5.1 Crossover in Meiosis

Meiosis begins with a single body cell. Each species has a different number of chromosomes, but to make this discussion easier, we will consider a cell with 2 chromosomes, one from the mother and one from the father. In preparation for meiosis, each chromosome duplicates and remains attached to its copy. This pair is known as a dyad or sister chromatids. Now, since the chromosomes were in pairs to begin with, each

dyad has a homologue, another dyad of the same shape and length. The dyads find their homologues and align with them lengthwise. The pair then join at one or more points so that a chromatid from one dyad becomes connected to a chromatid from its homologue (i.e. non-sister chromatids join). Subsequently, the dyads pull apart resulting in some chromatids having certain portions of their genes switched (Figure 6.1). This important event is known as crossover.

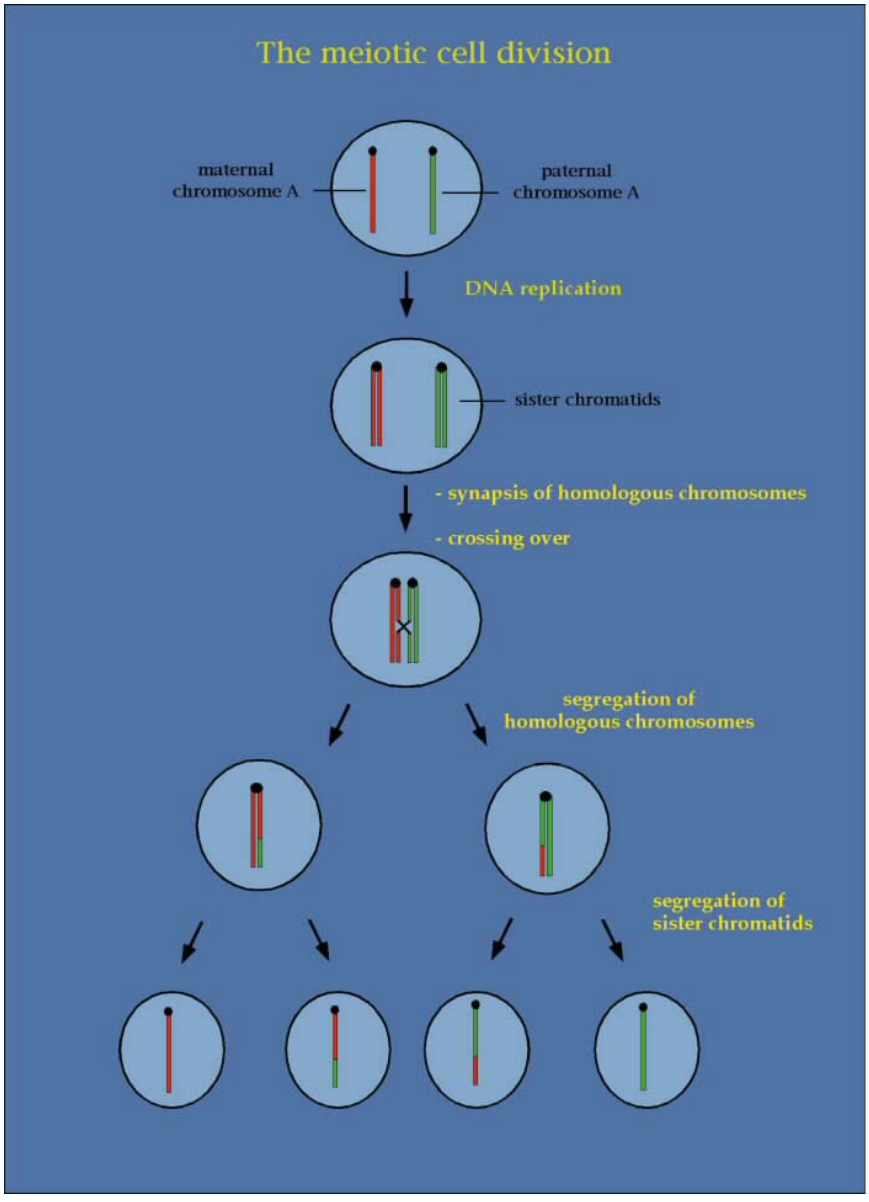

Figure 6.1: Crossover in meiotic cell division [4].

After crossing over, the dyads separate to opposite poles of the cell. This is followed by the first phase of meiotic division which produces two cells, each carrying a dyad. The chromatids then separate into independent chromosomes and migrate to opposite poles of the cells. In the second phase of meiotic division, the two new cells divide further, producing four cells in total, each containing a single chromosome. In this limited example, crossover results in two new chromosomes that are different from those of the initial cell. In general, each gamete will have a certain percentage of its chromosomes being different from those of either parent.

6.5.2 Mutation

Further variation in the offspring occurs when a gene mutates. Mutation refers to a random change at a locus on a chromosome. Point mutation can be the replacement of a single base by another. Deletion of a gene or part of it is another type of mutation. A third type is mutation by insertion of extra DNA into the gene. Finally, frame shift mutation results from the insertion or deletion of one or two genes followed by a misinterpretation of the position of all the genes that follow. Mutation occurs infrequently, thereby providing small changes that might be needed, while preventing a sudden, drastic perturbation in the population that could destabilize it.

6.6 Terminology

Terms from genetics frequently appear in GA literature, but the meaning can be quite different, often denoting in the latter a simpler interpretation of the same concept. The following list shows how these terms are used in the binary genetic algorithm.

Gene: A string of bits representing a parameter to be optimized. A gene is associated with a specific position in the chromosome.

Chromosome: Several genes concatenated together for optimization of multiple parameters.

Genotype: The uninstantiated chromosome; the chromosome as a set of variables.

Phenotype: The instantiated chromosome.

Alleles: The values that a gene can take.

Crossover: Taking two chromosomes and then switching a portion of each so that the first gets a piece of the second chromosome and vice versa.

Mutation: Flipping a single bit in the chromosome.

Individual: A single chromosome or its interpretation.

6.7 The Basic Genetic Algorithm

A typical genetic algorithm can be summarized in the following steps. In short, possible solutions are generated, tested, and, depending on the result of the test, are either discarded or used to form the next generation.

1. Choose parameters of the algorithm: population size, N, probability of crossover, p_c, and probability of mutation, p_m.
2. Identify an encoding for members of the solution space.
3. Randomly or heuristically generate an initial population of possible solutions.
4. Decode each individual.
5. Evaluate each individual's fitness.

6. Form a new population by selecting the fittest.
7. Introduce new members by performing crossover and mutation.
8. If termination condition is not met, go back to step 5.

A survey of the literature shows that genetic algorithms have been extensively applied to problems in both computer vision and software testing. Here, we apply genetic algorithms to the combined problem of testing computer vision software, and specifically to Phase 2 testing.

6.8 Genetic Algorithms and Testing

In testing for the purpose of triggering exceptions, one targets just those parts of a program, usually few, that could lead to hazardous conditions. As it turns out, genetic algorithms are particularly well-suited to this problem. A key element in a genetic algorithm is the fitness function that measures how close each potential solution is to the actual solution sought. In [48], Pargas *et al.* describe a genetic algorithm implementation for guiding test-data generation that can be customized to focus on various aspects of a software system. They specifically focus on statement and branch coverage.

In [73, 74], another application of genetic algorithms to test-data generation is described, with software safety as the focus. Statements in the software under test (SUT) that may cause hazards or raise exceptions are targeted. The fitness function is computed so that data that may cause a hazard or exception are considered fitter than those that are unlikely to have such an effect. Fitness is determined by the effect of the data on various critical points as it travels through the SUT. For example, suppose we are looking for data that will raise an exception near the end of the program. But early in the program, there is a checkpoint that causes the program to exit if the input values are less than zero.

Then the fitness function must assign a lower fitness to negative inputs and a higher fitness to positive inputs, because the negative inputs stand no chance of ever reaching the point where the exception can occur.

Many others have similarly reported on successful use of genetic algorithms in software testing. Work by Alander *et al.* [2] investigated how genetic algorithms might be applied to test data generation for black box testing. Specifically, a dynamic stress testing scenario was examined. Michael and McGraw describe GADGET [42], a tool that applies several optimization techniques to test data generation. Condition-decision coverage is formulated as function minimization. Genetic algorithms, simulated annealing, and gradient descent are compared and the standard genetic algorithm shows superior performance in simpler programs, while simulated annealing and gradient descent perform better in complex programs. In [43],Michael *et al.* propose a genetic algorithm approach to test data generation for general purpose software testing. The fitness criterion is condition-decision coverage, which requires both values of a conditional statement to be tested as well as each control statement. Harmanani and Karablieh [22] use a genetic algorithm to generate test patterns for testing digital circuits. A pattern's fitness is determined by the number of faults that pattern uncovers. DIGATE (Hsiao *et al.*, [25]) is a test generator for sequential circuits that also used genetic algorithms. Finally, similar to this thesis, Wang's PhD research studied the application of genetic algorithms to the problem of reliability in physical structures [78].

6.9 Genetic Algorithms in Computer Vision and Related Areas

Cinque *et al.* [12] propose a genetic algorithm for optimizing parameters for the segmentation of range images. Several algorithms with as many as eight parameters are

examined and ground truth image sets are used. The fitness criteria include how closely the segmented regions match the real regions and the gray level variance in each segmented region.

Da Silva *et al.* [17] apply a genetic algorithm to the stereo matching problem. (The matching or correspondence problem involves identifying points in two or more images that are projections of the same scene point. Once points are matched, depth information can be recovered.) Their genetic algorithm searches for the most likely corresponding point among many candidates.

Ahrens [1] proposes a genetic approach to superresolution, which is the process of creating a high resolution image from a collection of low resolution images. It is again a matching problem where one seeks the points in the images that are projections of the same scene points.

Lee and Bulitko [33] present a genetic approach to image interpretation via machine learning. They address the problem of the long computational times required to scan large operator libraries. They report that their genetic algorithm reduces the operator library needed to accurately interpret an image.

Zhang *et al.* [80] use a genetic algorithm to solve the matching problem in the registration of medical images. Image registration involves finding similarities between points in two images in order to align them where they overlap. Thus this problem is analogous to that of stereo matching, as is the manner in which a genetic algorithm can be applied to it.

6.10 A Genetic Algorithm Approach to Color Tolerance Search

We implement our solution with the help of GA Playground [18], a generalized genetic algorithm tool that is customizable in various ways. Written in Java, it is designed to allow the definition of new problems along with fitness functions for them. Defining a new problem involves editing a text file containing a list of attribute-value pairs grouped by data type, similar to a windows .ini file (Figure 6.2 shows an example). The fitness function is specified by modifying just one Java class to include a case statement for the new problem. The value returned by this case statement is passed to GA Playground, which further evaluates it according to the value of the MinMax field in the problem definition file. MinMax = 1 denotes a maximization problem. MinMax = 2 denotes a minimization problem. In the latter case, GA Playground regards returned values as improving when they decrease.

```
[Integers]
Problem Code=30
Population Size=30
Number of Genes=10
Map Order=0
Def Order=0
GA Type=1
MinMax Type=2
Crossover Type=1
Mutation Type=1
Selection Type=1
Inversion Type=1
Stagnation Limit=3
Degrade Limit=4
Survivors Percent=20
Redundancy Factor=9
Number of Variables=10
User Defined Integer=1

[Reals]
Crossover Rate=1
Mutation Rate=0.1
Inversion Rate=0
Shuffle Rate=0.5
Inversion Shuffle=0
Kick Distribution=0.9
Exit Value=0
Exit Tolerance=0.001
Min Value=0
Max Value=400
Step Value=1
Default Value=1
Kin Competition Factor=0.8
User Defined Real=0

[Strings]
Title=A two variable minimization problem for image analysis.
Description= min(abs(num regions found - num actual regions))
Map Delimiter=,
Input String #1=None
Input String #2=None
User Defined Expression=None
User Defined String=barcode.bmp,255,133,196

[Flags]
Status Help=True
Text Window=False
Graphic Window=True
Sound=False
Logging=False
User Defined Flag=False
```

Figure 6.2: A problem definition file for use in GA Playground. Once loaded, these fields can be adjusted dynamically to fine-tune the GA's performance [18].

6.10.1 Representation:

While many different representations have been devised and found useful, in the standard genetic algorithm binary strings are used to encode possible solutions. For the YIQ model, the parameters are the T_y and T_{iq}, both integers. For the HSV model, there are three parameters, T_h, T_s, and T_v, also integers. The most natural representation for these is the concatenation of a binary encoding of their values, so in our case we conform to tradition. For k parameters, each would be converted to its binary form and then all k strings would be concatenated together into one binary string. For example, since Y can go from 0 to 255 and T_y is the upper bound for the absolute difference between two values of Y, the range is the same for T_y. Therefore we need only 8 bits to represent T_y. No more than 9 bits are required to represent T_{iq}. Given two pixels, P_1 and P_2, the parameter T_{iq} is used to assess the sum of square differences of their two (Q,T) pairs. If $((I_1 - I_2)^2 + (Q_1 - Q_2)^2) \leq T_{iq}$ then P_1 and P_2 are deemed to be of the same color; otherwise they are different colors. In segmentation trials, we observed that values of T_{iq} beyond 500 tended to over-segment. Therefore, allowing for some flexibility, we set an upper bound of 400 for this parameter (specified in the Max Value field under Reals in the problem definition file), thus requiring 9 bits to represent it. In total 17 bits would suffice to represent the parameters for the YIQ model. However, in order to accommodate a large number of problems with a wide variety of range requirements, GA Playground provides equal length encodings that can cover many more values than needed in practice. In GA Playground, all chromosomes are in the form of a Unicode character, which is 16 bits long. This can encode 2^{16} values for each parameter, which should cover

the needs of most optimization problems.

6.10.2 Initial Population:

The population size can be set in the problem definition file (see Figure 6.2). A population size of 30 has worked well in experiments. This value is specified in the field Population Size in the problem definition file. GA Playground randomly generates 30 binary strings that will make up the initial population.

6.10.3 Fitness:

The fitness of an individual representing x is the value of $f(x)$ where in our case f is a measure of how well parameters perform when used in selected image analysis tasks. We seek to minimize f. In general, the fitness of an individual is a determination of how close it is to the optimal solution. Our problem is made easier by the fact that we do not seek the optimal set of parameters but rather a "good enough" set. For example, a set of parameters that finds 90% of a tag would be good enough if we assume that percentage includes all the information essential for reading the tag. This is not an unreasonable assumption; the color barcode, for instance, can have a wide margin.

For our color visual tags a number of options are available for evaluating the fitness of parameters. These involve comparisons between values calculated through image analysis and known values for the same image set, and they include the following.

- The number of bounding rectangles compared to the number known to be present.
- The raw number of pixels found compared to the number estimated or known to be present. If the entire image is occupied by a tag, for example, then the correct number of pixels is simply the length times the width of the image. Alternatively,

if the bounding rectangle is specified, the correct pixel count can in this case also be determined exactly.

- The placement of the corners of bounding boxes compared to their known positions. This can be determined by finding the absolute difference in the minimum rows, maximum rows, minimum columns, and maximum columns. This gives four comparison values which can be combined and compared using the sum of square differences method (SSD).
- Hamming distance between the tag value read and the correct value previously determined (tag values are easy to determine by manual inspection). We observed, for example, that a poor reading often resulted in a barcode representing 4 being read as a zero. This seemed like a big difference until we realized that there is only a Hamming distance of 1 between 0100 and 0000. This led to the choice of Hamming distance as a measure of fitness rather than the decimal difference. This makes even more sense when we note that the barcodes are read as binary numbers that are then converted to decimal for output.
- The ratio of the number of pixels of a given color to the total number of pixels. For example in the SB tags, the black patch occupies a certain known percentage of the tag. The better tolerances will yield a ratio close to this percentage.

Here are the steps we follow to compute *f*.

1. Decode chromosome, $d(C) = (T_y, T_{iq})$
2. Use (T_y, T_{iq}) in a call to an image analysis function with a selected image as input. For example, to use the Hamming distance approach, we would proceed as follows.

3. Read(tag, color, T_y, T_{iq}) = v_1
4. Calculate the Hamming distance, $H(v_1, v_2)$, between the value just read, v_1, and the correct value, v_2.
5. Pass $H(v_1, v_2)$ to GA Playground for further computation.

6.10.4 Selection:

Selection is performed using a Monte Carlo simulation of a roulette wheel. In a Monte Carlo simulation, suppose there are three events, E_1, E_2, E_3, with the following probability distribution: $p(E_1)=½$, $p(E_2)=¼$, $p(E_3)=¼$. We first compute the cumulative distribution function: $q(E_1)=½$, $q(E_2)= ¾$, $q(E_3)=1$. We can simulate an occurrence of these events by repeatedly generating a random number $r \in [0..1]$. If $r \leq ½$, we say that E_1 has occurred. If $½ < r \leq ¾$, we say that E_2 has occurred. Finally, if $¾ < r \leq 1$, we say that E_3 has occurred. Similarly, the roulette wheel for natural selection is constructed as follows.

1. Calculate the fitness for each chromosome, $f(C_i)$.
2. Calculate the total fitness, F by taking the sum of all the terms in 1.
3. Calculate the probability of selection of each chromosome, $p_i = f(C_i)/F$.
4. Calculate a cumulative probability q_i for each chromosome C_i by computing the sum up to the ith term in 3.

Now we simulate spinning the wheel N times, where N is the chosen population size, in this case 30. We do this in two steps:

1. Generate a random real number r, $0 \leq r \leq 1$.
2. Select a chromosome, C_i, if $q_{i-1} < r \leq q_i$.

6.10.5 Applying Genetic Operators.

We apply single-point crossover with initial probability $p_c = 0.25$, and bit mutation with initial probability $p_m = 0.01$. Both these parameters can be adjusted offline in the problem definition file as well as online in GA Playground. We first apply crossover. We generate a random number $r \in [0..1]$. If $r < p_c$, the given chromosome is selected for crossover. This is repeated for all the chromosomes in the population so that each has a chance of crossover. The crossover itself, also known as mating, is performed by first choosing a random number between 1 and the length of the chromosome. This number marks the crossover point. We take pairs of the chromosomes that are slated for crossover, and we switch the portions after the crossover point. For example, let 0101010000101100101 and 1010011101101010 be two chromosomes and let 14 be the crossover point. Then upon applying crossover, the two are replaced by 0101010000101101010 and 1010011101101000101.

After crossover, every bit in every chromosome is considered for mutation, which, when it occurs, inverts the bit. Again a random number is generated. If $r < p_m$, the given bit is mutated. For example, if the 3rd and 6th bits in 1101010011100101111 are mutated, this chromosome is replaced by 1111000011100101111.

The situation for the HSV model is completely analogous with a few modifications. There are three parameters to optimize, namely H tolerance, S threshold, and V tolerance. We shall call them T_h, T_s, and T_v. The range for H is [0..360], and it is [0..255] for S and V. Allowing 100 different values for each parameter is reasonable and can be encoded with $8 \times 3 = 24$ bit chromosomes. Once again the representation provided in GA Playground, which gives $16 \times 3 = 48$ bits (3 Unicode characters), is more

than sufficient for this set of parameters.

While experiments are used to determine what crossover and mutation probabilities work best, they are typically small, say, 0.25 for crossover and 0.01 for mutation. However, genetic algorithm variations exist in which these probabilities are much higher. In our experiments we adjusted these values and examined how the changes affected the performance of the GA in terms of the quality of the optimization and computational efficiency. No significant changes were observed. In other studies these probabilities are fixed at the beginning and are gradually raised or lowered, depending on how well the operator concerned is performing [15]. The expected number of chromosomes that undergo crossover is $N p_c$ and that of mutated bits is $Nl p_m$, where l is the number of bits in a chromosome.

This, then, is one full cycle of the genetic algorithm. Notice how completely generic crossover and mutation are—we make no reference at all to the underlying meaning of the chromosome, which is what makes it possible to use this exact same procedure for all the different fitness functions in our application. After the genetic operators have been applied, the current population undergoes fitness evaluation and the cycle repeats. The number of iterations, usually set in advance, depends on the population size and the computing resources available. Alternatively, a genetic algorithm can be set to terminate when a sufficiently fit individual has been obtained or when no further improvements in fitness are seen. In all test runs in out experiments, a good set of parameters is produced in less than 100 iterations. This is important for fitness functions such as ours that involve potentially time-consuming image analysis tasks.

As in the example above, the original genetic algorithm used binary encoding,

single-point crossover, and bit mutation. While some researchers consider these to be essential elements in a true genetic algorithm, many variations have been introduced with competitive results. For example, experiments have been done with real representation, rather than binary, and the results have been good for certain problems for which such representation has been deemed more suitable. In the optimization of certain real-valued functions, for example, some researchers have found better results by using a population of real numbers. Of course, the crossover and mutation operators have to be modified to suit each new representation. For example, to produce a child from two real numbers, one might calculate their average. Despite their success, there has been concern that such modifications produce tools that lack the traditional genetic algorithm's theoretical backing, but this theory itself is not well understood (e.g. see schema discussion in [44]).

6.11 Experiments with GA Playground

In this section we discuss more details of our implementation of the five fitness functions outlined earlier in this chapter. These functions are encapsulated in the module Tracey.java, which derives its name from its beginnings as a class for contour tracing. While it has evolved beyond this purpose, most of the new functionality relies on contour tracing as a primary task to initially find where tags are located in an image. For example, to count pixels, the contour around a tag of a given color is traced. Then from the resulting chain code, the bounding box is computed and used as input to the `countPixels` function. How well `traceCountour` performs is, therefore, a good indicator of the overall robustness of the application. Tracey replicates exactly the behavior of the corresponding functions in the Anna Xtra. This is important to ensure that the behavior tested does not differ from the potential behavior of the system in actual

operation.

Note also that although the chain code represents a tighter and more accurate boundary definition for the tags, bounding rectangles provide a more compact representation that is easier to work with. Furthermore, in many of the educational applications that we have considered, there is no danger of overlap between bounding boxes since tags are located in the middle of objects with plenty of space left between tags in the captured image. Should this issue become a concern (e.g. in the orientation tags where the spots are close together), the chain code can be used instead, with appropriate code revision.

6.11.1 Number of Bounding Rectangles

In this case a call is made to `Tracey.getBoundingRecs`. This function requires four arguments and returns the number of bounding rectangles found. The first argument is a 3-element integer array containing the r, g, b values of the target color. The second argument is an array for storing the list of bounding rectangles found. This can be passed in as null if the rectangles are not needed, as is the case here where we are only interested in the number of rectangles found, not the rectangles themselves. The third and fourth arguments are the tolerances, which will be provided by the GA.

In addition, a file name is required specifying the image file containing the test image. The .par file (see Figure 6.2) provides a number of fields for discretionary use. The User Defined String field is particularly useful. It provides a means to specify arguments of any type as strings, which can then be parsed later into their appropriate types. In this case, this field holds the following string:

"<file_name>,<r>,<g>,<b>"

Lastly, the correct number of rectangles (to be compared with the number returned) is stored in the field User Defined Integer in the .par file. Again, all the values in the .par file are available for modification while the program is running.

The objective function, which the GA is charged with minimizing, computes the absolute difference between the value stored in the User Defined Integer field and the number returned by `getBoundingRecs`.

6.11.2 Positions of Bounding Rectangles

This case builds on the first case. Upon finding the correct number of bounding rectangles, it may be desirable to subject the parameters to further testing, namely, whether the rectangles found are in the correct locations. It is possible that the number being correct is merely coincidental, and that it in fact counts a different set of rectangles from that desired. The actual list of rectangles found is returned in the second (non-null) argument to getBoundingRecs. The User Defined String holds the string

"<file_name>,<r>,<g>,<b>,<x_{11} x_{21} y_{11} y_{21}>,…, < $x_{1\,m}$ $x_{2\,m}$ $y_{1\,m}$ $y_{2\,m}$ >".

Here x_1 and x_2 are the minimum and maximum columns and y_1 and y_2 are the minimum and maximum rows, respectively, and there are m tags of the given color in the image. The GA tries to minimize the function $1/m\sum\sqrt{((x_{1i}-x_{2i})^2 + (y_{1i}-y_{2i})^2)}$.

6.11.3 Area Occupied by a Tag

This simple, yet effective, case tests through counting pixels. It invokes the `Tracey.countPixels` function, which takes 4 arguments, of which the last two (three for the HSV model) are the tolerances as usual. The first argument is again the target color and the second is a 4-element integer array representing the known bounding

rectangle for the tag. A separate module can be used that records the bounding rectangle when a user clicks at the corners of a tag or performs a selection of the tag in a displayed image, assuming tags are rectangular. Otherwise the area inside the selection that is not part of the tag must be accounted for in the minimization. As the name indicates `countPixels` returns the number of pixels of the target color found within the bounding box. The GA is asked to minimize the absolute difference between this value and the actual area of the bounding box, which is simply $(x_2-x_1)\times(y_2-y_1)$, again assuming rectangular tags. Where the tag does not occupy the entire bounding rectangle, and it is not known precisely how much of the rectangle it occupies, a tolerance value as large as needed can be provided in the .par file's Exit Value Tolerance field. Indeed, even when the tag is rectangular and fills the bounding box, a set of parameters that find, say, 90% of it, can be good enough if the important portion of the tag that actually encodes a value is in that 90%. Minimization down to zero is not necessary or even desirable in this case.

6.11.4 Proportion of Color

This test builds on the area-based test but looks at the proportion of pixels in the tag that are of the target color. This test is suitable for use with single bit (SB) tags. The percentage of the tag occupied by the patch is assumed known and is then provided in the User Defined Real field of the .par file. The GA minimizes the absolute difference between this value and that computed by dividing by the tag size the result returned by countPixels. As in the area-based case, the minimization should stop when the objective function produces a small enough result, not necessarily zero.

6.11.5 Hamming Distance

In the Hamming distance test, the particular method for reading each tag has to be invoked. In these experiments, it has only been tested with the color barcode. A call is made to `Tracey.readBarCode`. Besides the tolerances, `readBarcode` takes a color and a bounding rectangle, reads the numerical value represented there and returns it. The GA minimizes the Hamming distance between the value returned by `readBarCode` and the correct value provided in the .par file's User Defined Integer field. This value is easy to determine by manual inspection of the tag. The User Defined String contains a string of the form "<file_name>,<r>,<g>,<b>,< x_1 x_2 y_1 y_2>".

6.12 Experimental Results

6.12.1 Experiment 1

To compare the two parameter approximation approaches, we had each approach generate tolerances, which we then used to segment images. The results shown in table 6.1 were obtained from the image shown in figure 6.3. It was not necessary to perform actual segmentation. Instead, segmentation was simulated using `countPixels`, thus, providing a high-precision measure of each method's performance.

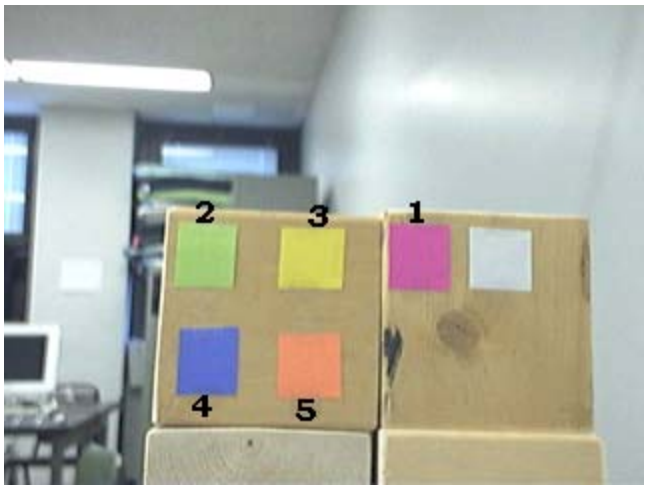

Figure 6.3: Single bit tags, with value 1, used to test parameters obtained by the MD and GA methods.

Table 6.2: Comparison by Segmentation

Color	RGB	Rect (x,y,X,Y)	MD Tols	MD Seg	GA Tols	GA Seg
1	202, 98, 166	193,109,222,142	8, 42	86.1%	328, 366	97.0%
2	170, 209, 125	87,109,114,141	8, 81	86.3%	87, 493	99.3%
3	215, 213, 96	138,111,169,142	6, 27	74.3%	387, 489	98.0%
4	97, 113, 204	86,161,116,195	10, 156	82.7%	359, 474	89.0%
5	236, 150, 110	136,163,167,195	6, 41	76.2%	83, 488	96.5%

Average Segmentation for MD: 81.1%

Average Segmentation for GA: 96.0%

The columns MD Tols and GA Tols show the Y and IQ tolerance pairs obtained by each method. The column MD Seg shows the segmentation percentages obtained using tolerances generated by the MD method. The GA Seg column shows the

corresponding results for the GA approach. The averages are also calculated. Segmentation using the GA tolerances is better by 15 percentage points. However, while the MD approach may not produce the best tolerances, it reveals useful properties of the image, which the GA, being a blind approach, does not. For example, the Y tolerance values in the MD Tols column show that brightness varies little across colors and from pixel to pixel.

6.12.2 Experiment 2

The image shown in figure 4.6 in Chapter 4 is used in this experiment. Parameters obtained by the GA and MD approaches are applied 9 times to different pixel samples (i.e. each tag is sampled three times). For the GA the fitness function used is `readBarCode`. In this function, the Hamming distance between the value read and the actual known value of the barcode is minimized. Since the parameters obtained this way are used to perform the same operation with which they were tested, they perform very well. Table 6.2 below compares the results of the GA approach to the MD approach. MD performs poorly in the last two tests. In fairness, it should be mentioned that the image used (see figure 4.6) shows very poor quality in the region occupied by the last tag (blue). The best quality portion of the image (first tag) is read correctly two out of three times, and is off by only 1 bit the other time. As much as possible, therefore, images should be obtained in a manner that maximizes quality.

Table 6.3: Comparison by Value Read and Hamming Distance

Barcode value	Value read—MD	Hamming Dist—MD	Vaue read—GA	Hamming Dist—GA
0	0	0	0	0
0	1	1	0	0
1	0	1	1	0
0	0	0	0	0
0	8	1	0	0
1	5	1	1	0
0	4	1	0	0
0	14	3	0	0
1	8	2	1	0

6.12.3 Experiment 3

In this experiment, sixty photographs of tags were taken. For each of three tag types, the same set of tagged objects was rearranged and captured five times in each of four scenarios. These scenarios were created by changing the environment in a number of ways, as follows.

1. Bright light, direct view
2. Bright light, a non-reflecting plastic surface
3. Dim light, direct view
4. Dim light, a non-reflecting plastic surface

The purpose of the experiment was to assess more thoroughly the effects of 1) different levels of lighting and 2) viewing through a non-reflecting surface versus viewing directly. Both Plexiglas and plastic were used as non-reflecting surfaces. Each image contained four tags in four different colors, on average. The tags were photographed using a medium-resolution, low-cost webcam, which, as mentioned in Chapter 4, gives more accurate readings than a high resolution camera.

Of the sixty photographs available, ten were eliminated because they were very similar to other pictures, so that fifty images were actually used in the testing. For each scenario and for each of three tag types (SB, PH, tricolor barcode), an image was selected to be used with the genetic algorithm to generate parameters for both the YIQ and HSV models (this is 24% of the test images). The number of bounding rectangles found was used to determine fitness. Since there was only one tag of each color per image, a perfect reading would return a value of 1. Counting the number of bounding rectangles has previously been shown to be a good indicator of accuracy. The parameters returned by the GA were then used in finding the number of bounding rectangles for each tag in all 50 images. Since on average, there were four tags in each image, the parameters were tested in about 200 trials.

The following tables (6.4-6.6) show the results of the average number of bounding rectangles found for each tag in each scenario. The smaller this number is, the greater the reading accuracy.

Table 6.4: SB Tag Identification Accuracy

	Bounding boxes per tag—YIQ	**Bounding boxes per tag—HSV**
Bright, direct	2.3	1.8
Bright, indirect	3.2	3.0
Dim - direct	2.0	1.5
Dim - indirect	3.5	3.1

Table 6.5: PH Tag Identification Accuracy

	Bounding boxes per tag—YIQ	**Bounding boxes per tag—HSV**
Bright, direct	2.5	1.8
Bright, indirect	3.2	3.0
Dim - direct	1.9	1.8
Dim - indirect	3.4	3.1

Table 6.5: Tricolor Tag Identification Accuracy

	Bounding boxes per tag—YIQ	**Bounding boxes per tag—HSV**
Bright, direct	2.5	2.1
Bright, indirect	3.3	3.1
Dim - direct	2.0	2.0
Dim - indirect	3.5	3.2

The results show that direct viewing in dim light provides the most accurate readings of color visual tags. Overall, with direct or indirect viewing, the HSV model gives superior performance in all scenarios. When a surface is required, which is the

more likely scenario for the types of collaborative educational activities we had in mind, then the HSV model should be used in bright light.

With direct viewing, on the other hand, bright light reflects off the tags and causes colors to be captured incorrectly. Thus dimmer light is recommended for direct viewing. For very poor quality images, the genetic algorithm did not converge quickly and was stopped at 10 generations. This was necessary because of the long running times of the image analysis functions, especially for larger images. Images larger than 320 by 240 are not recommended for use with the GA.

6.13 Listing of Tracey.java

The entire Tracey class is presented here. It contains the methods that are invoked during fitness computation, namely `getBoundingRecs`, `countPixels`, and `readBarCode`.

```
import java.awt.image.SampleModel;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import javax.media.jai.TiledImage;
import java.awt.image.Raster;
import java.awt.image.WritableRaster;
import java.util.ArrayList;

public class Tracey
{
	// data
	private static PlanarImage pi;
	private static WritableRaster wr;
	private static int width;
	private static int height;
	private static boolean goodImgCopy;
	private static int[] chainCode;
	private static int chainLen;
	private final static int MAXCHAINLEN = 100000;

	// Constructor
	public Tracey(String imgFilename)
	{
		pi = JAI.create("fileload", imgFilename);
		wr = Raster.createWritableRaster
			(pi.getSampleModel(), null);
		width = pi.getWidth();
		height = pi.getHeight();
		goodImgCopy = false;
		chainLen = 0;
		chainCode = new int[MAXCHAINLEN];
	}

	// Empty constructor
	public Tracey()
	{
		pi = JAI.create("fileload", "zero.bmp");
		wr = Raster.createWritableRaster
			(pi.getSampleModel(), null);
		width = pi.getWidth();
		height = pi.getHeight();
		goodImgCopy = false;
		chainLen = 0;
```

```
      chainCode = new int[MAXCHAINLEN];
}

// readBarCode
// (YIQ)
// ---------------------------------------------------------
public static int readBarCode(int[] rgb, int[] bBox,
      int yTol, int iqTol)
{
      int t = 0;        // count of color transitions
      int bars = 0;     // number of bars
      int[] currentRGB = {0,0,0};
      int h = 0, w = 0;             // image indices;
      int intCode = 0;
      int mask = 8;
      int[] colorOne = {0,0,0};
      int dir = 0, dw = 0, dh = 0;
      int stopw = 0, stoph = 0;
      int[] diag1 = {0,0,0};
      int[] diag2 = {0,0,0};
      int minCol=bBox[0], maxCol=bBox[1];
      int minRow=bBox[2], maxRow=bBox[3];
      int[] pixel = {0,0,0};


      if (!goodImgCopy)
      {
            wr = pi.copyData();
            goodImgCopy = true;
      }

      // calculate coefficients of diagonals
      /* diag1 is / and diag2 is \ */
      calcLineCoeffs(minCol,maxRow,maxCol,minRow, diag1);
      calcLineCoeffs(minCol,minRow,maxCol,maxRow, diag2);

      currentRGB[0] = rgb[0];
      currentRGB[1] = rgb[1];
      currentRGB[2] = rgb[2];


      for(dir=1;dir<5;dir++)
      {
            t = 0;
            intCode = 0;
            switch(dir)
            {
                  case 1:           // horizontal
                        dw = 1;
                        dh = 0;
                        h = (maxRow + minRow)/2;
                        w = minCol;
                        stopw = maxCol;
                        stoph = (maxRow + minRow)/2;
                        break;

                  case 2:           // vertical
```

```
                dw = 0;
                dh = 1;
                h = minRow;
                w = (maxCol + minCol)/2;
                stopw = (maxCol + minCol)/2;
                stoph = maxRow;
                break;

        case 3:           // bottom left to top right
                dw = 1;
                h = maxRow;
                w = minCol;
                stopw = maxCol;
                stoph = minRow;
                t = -2;
                break;

        case 4:           // top left to bottom right
                dw = 1;
                h = minRow;
                w = minCol;
                stopw = maxCol;
                stoph = maxRow;
                t = -2;
                break;
}

while (true)
{
        if((dir==1)&&(w==stopw))
                break;
        if((dir==2)&&(h==stoph))
                break;
        if((dir==3)&&((w==stopw)||(h==stoph)))
                break;
        if((dir==4)&&((w==stopw)||(h==stoph)))
                break;

        wr.getPixel(w,h,pixel);

        if(!compareYIQ(pixel, currentRGB, yTol,
                iqTol))
        {
                t++;

                if(t==1)
                {
                        colorOne[0] = pixel[0];
                        colorOne[1] = pixel[1];
                        colorOne[2] = pixel[2];
                }

                if ((t%2)==0)
                {
                        bars = t/2;
                        if(compareYIQ(currentRGB,
                                colorOne, yTol, iqTol)
```

```
                                    && (bars>1) && (bars<=5))
                                    intCode |= (mask >>>
                                          (bars-2));
                              }
                              currentRGB[0] = pixel[0];
                              currentRGB[1] = pixel[1];
                              currentRGB[2] = pixel[2];
                        }
                        if((dir==1)||(dir==2))
                        {
                              w += dw;
                              h += dh;
                        }
                        else if(dir==3)
                        {
                              w+=dw;
                              h = (w * diag1[1] + diag1[2])/diag1[0];
                        }
                        else if(dir==4)
                        {
                              w+=dw;
                              h = (w * diag2[1] + diag2[2])/diag2[0];
                        }
                  }
                  if (t>=20) break; // Success!
            }
            if (t<20)
                  intCode = -1;
            return intCode;
      }


      // getBoundingRecs
      public static int getBoundingRecs(int[] rgb,
            ArrayList recList, int yTol, int iqTol)
      {
            int[] loc = {0,0};
            int[] bBox = {10000,-1,10000,-1};
            int recs = 0;
            StringBuffer stringBox = new StringBuffer();

            //while(true)
            while(recs < 100)
            {
                  if(findFirst(loc, rgb, yTol, iqTol))
                  {
                        recs++;
                        traceContour(loc, rgb, yTol, iqTol);
                        getBoundingBox(bBox);
                        deleteBox(bBox);
                        if(recList != null)
                        {
                              for (int i=0;i<4;i++)
                              {
                                    stringBox.append(bBox[i]);
                                    stringBox.append(" ");
                              }
```

```
                        stringBox.deleteCharAt
                              (stringBox.length() - 1);
                        recList.add(stringBox.toString());
                        stringBox.delete(0,stringBox.length());
                  }
            }
            else
            {
                  break;
            }
      }
      if(recs>0)
            goodImgCopy = false;
      return recs;
}

// findFirst
private static boolean findFirst(int[] loc,
      int[] inputPixel, int yTol, int iqTol)
{
      boolean found = false;
      int[] pixel = {0,0,0};

      if (!goodImgCopy)
      {
            wr = pi.copyData();
            goodImgCopy = true;
      }

      for(int w=0;w<width;w++)
      {
            for(int h=0;h<height;h++)
            {
                  wr.getPixel(w,h,pixel);
                  if(compareYIQ(pixel, inputPixel, yTol,
                        iqTol))
                  {
                        found = true;
                        loc[0] = w;
                        loc[1] = h;
                        break;
                  }
            }
            if(found) break;
      }
      if (!found)
      {
            loc[0] = -1;
            loc[1] = -1;
      }
      return found;
}

// traceContour
private static void traceContour(int[] startLoc,
      int[] rgb, int yTol, int iqTol)
{
```

```
int count = 3;
boolean first = false;
int searchDir = 0;
int[] currentLoc = {0,0};

chainCode[0] = startLoc[0];
chainCode[1] = startLoc[1];
currentLoc[0] = startLoc[0];
currentLoc[1] = startLoc[1];
searchDir = 6;
first = true;

int index = 2;
int[] loc1 = {0,0};
int[] loc2 = {0,0};
int[] loc3 = {0,0};

while (((startLoc[0] != currentLoc[0])||
      (startLoc[1] != currentLoc[1])) || first)
{
      boolean found = false;
      while (!found)
      {
            count--;
            if (count==0) break;
            getNeighbor(loc1, currentLoc,
                  ((searchDir - 1 + 8) % 8));
            getNeighbor(loc2, currentLoc, searchDir);
            getNeighbor(loc3, currentLoc,
                  ((searchDir + 1 + 8) % 8));
            if (isInR(loc1, rgb, yTol, iqTol))
            {
                  currentLoc[0] = loc1[0];
                  currentLoc[1] = loc1[1];
                  found = true;
                  if (index < MAXCHAINLEN)
                  {
                        chainCode[index] =
                              (searchDir - 1 + 8) % 8;
                        index++;
                  }
                  searchDir = (searchDir - 2 + 8) % 8;
            }
            else if (isInR(loc2, rgb, yTol, iqTol))
            {
                  currentLoc[0] = loc2[0];
                  currentLoc[1] = loc2[1];
                  found = true;
                  if (index < MAXCHAINLEN)
                  {
                        chainCode[index] = searchDir;
                        index++;
                  }
            }
            else if (isInR(loc3, rgb, yTol, iqTol))
            {
                  currentLoc[0] = loc3[0];
```

```
                        currentLoc[1] = loc3[1];
                        found = true;
                        if (index < MAXCHAINLEN)
                        {
                              chainCode[index] =
                                    (searchDir + 1 + 8) % 8;
                              index++;
                        }
                  }
                  else
                        searchDir = (searchDir + 2 + 8) % 8;
            }
            first = false;
      }
      chainLen = index;
}

// getNeighbor
private static boolean getNeighbor(int[]loc,
      int[] currLoc, int dir)
{
      boolean validDir = true;

      switch(dir)
      {
            case 0:
                  loc[0] = currLoc[0] + 1;
                  loc[1] = currLoc[1];
                  break;
            case 1:
                  loc[0] = currLoc[0] + 1;
                  loc[1] = currLoc[1] - 1;
                  break;
            case 2:
                  loc[0] = currLoc[0];
                  loc[1] = currLoc[1] - 1;
                  break;
            case 3:
                  loc[0] = currLoc[0] - 1;
                  loc[1] = currLoc[1] - 1;
                  break;
            case 4:
                  loc[0] = currLoc[0] - 1;
                  loc[1] = currLoc[1];
                  break;
            case 5:
                  loc[0] = currLoc[0] - 1;
                  loc[1] = currLoc[1] + 1;
                  break;
            case 6:
                  loc[0] = currLoc[0];
                  loc[1] = currLoc[1] + 1;
                  break;
            case 7:
                  loc[0] = currLoc[0] + 1;
                  loc[1] = currLoc[1] + 1 ;
                  break;
```

```
            default:    // invalid direction
                  validDir = false;
                  break;
      }
      return validDir;
}

// isInR
private static boolean isInR(int[] loc,
      int[] inputRGB, int yTol, int iqTol)
{
      int[] pixel = {0,0,0};
      boolean ok = false;
      if((loc[0]>=0)&&(loc[1]>=0)&&
            (loc[0]<width)&&(loc[1]<height))
      {
            wr.getPixel(loc[0], loc[1], pixel);
            ok = compareYIQ(pixel, inputRGB, yTol, iqTol);
      }
      return ok;
}

// compareYIQ
private static boolean compareYIQ(int[] thisPixel,
      int[] inputPixel, int yTol, int iqTol)
{
      int y1 = yValue(thisPixel);
      int i1 = iValue(thisPixel);
      int q1 = qValue(thisPixel);

      int y2 = yValue(inputPixel);
      int i2 = iValue(inputPixel);
      int q2 = qValue(inputPixel);

      int yDiff = (y1<y2) ? (y2 - y1) : (y1 - y2);
      int iqDiff = (i2-i1)*(i2-i1) + (q2-q1)*(q2-q1);

      return ((yDiff < yTol) && (iqDiff < iqTol));
}

// yValue
private static int yValue(int[] rgb)
{
      return ((( rgb[0] * 299 ) + ( rgb[1] * 587 ) +
            ( rgb[2] * 114 )) / 1000 );
}

// iValue
private static int iValue(int[] rgb)
{
      return ((( rgb[0] * 596 ) - ( rgb[1] * 275 ) -
            ( rgb[2] * 321 )) / 1000 );
}

// qValue
private static int qValue(int[] rgb)
{
```

```
      return ((( rgb[0] * 212 ) - ( rgb[1] * 523 ) +
            ( rgb[2] * 311 )) / 1000 );
}

// getBoundingBox
private static void getBoundingBox(int[] bBox)
{
      int k;
      int[] loc = {0,0};
      int[] tempLoc = {0,0};
      loc[0] = chainCode[0];
      loc[1] = chainCode[1];

      // make sure bBox has the right initial values
      bBox[0] = 10000;
      bBox[1] = -1;
      bBox[2] = 10000;
      bBox[3] = -1;

      if (chainLen == 2)      // isolated pixel
      {
            bBox[0] = chainCode[0];
            bBox[1] = chainCode[0];
            bBox[2] = chainCode[1];
            bBox[3] = chainCode[1];
      }
      if (chainLen > 2)
      {
            for(k=2; k<chainLen; k++)
            {
                  getNeighbor(tempLoc, loc, chainCode[k]);
                  if (tempLoc[0] < bBox[0]) bBox[0] = tempLoc[0];
                  if (tempLoc[0] > bBox[1]) bBox[1] = tempLoc[0];
                  if (tempLoc[1] < bBox[2]) bBox[2] = tempLoc[1];
                  if (tempLoc[1] > bBox[3]) bBox[3] = tempLoc[1];
                  loc[0] = tempLoc[0];
                  loc[1] = tempLoc[1];
            }
      }
}

// deleteBox
private static void deleteBox(int[] bBox)
{
      int w, h;
      int[] blackPixel = {0,0,0};
      for (w=bBox[0];w<=bBox[1];w++)
            for(h=bBox[2];h<=bBox[3];h++)
            {
                  wr.setPixel(w,h,blackPixel);
            }
}

// countPixels
// count pixles of given color within given bounding box
// -----------------------------------------------------
public static int countPixels(int[] rgb, int[] bBox,
```

```
            int yTol, int iqTol)
      {
            int w, h, pixelCount = 0;
            int[] thisRGB = {0,0,0};

            if (!goodImgCopy)
            {
                  wr = pi.copyData();
                  goodImgCopy = true;
            }

            for(w=bBox[0];w<bBox[1];w++)
                  for(h=bBox[2];h<bBox[3];h++)
                  {
                        // retrieve current r, g, b
                        wr.getPixel(w, h, thisRGB);

                        // compare current r,g,b to input R,G,B
                        if (compareYIQ(thisRGB, rgb, yTol, iqTol))
                        {
                              pixelCount++;
                        }
                  }
            return pixelCount;
      }

      // calcLineCoeffs
      // calculate the coefficients a, b, c in ay=bx+c,
      // the line defined by the points (y1,x1) and (y2,x2)
      private static void calcLineCoeffs(int x1, int y1,
            int x2, int y2, int c[])
      {
            c[0] = x2-x1;
            c[1] = y2-y1;
            c[2] = y1*x2 - x1*y2;
      }


}
```

CHAPTER 7.

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This dissertation examines the problem of uncertain behavior in interactive systems that are based on computer vision. The main source of this problem is found to be uncertainty in the image input data. While various techniques exist for improving the quality of captured images before attempting to interpret them, none can completely eliminate their unpredictability, as this mainly arises from environmental factors beyond our control. This means that several image values, or pixel values, are going to have the same meaning and will need to be interpreted as such. The question, then, is how to determine this range of values that map into the same interpretation. In this research, this issue is looked at as a parameter optimization problem and also as a testing problem. This has made it possible to take advantage of techniques from software testing as well as optimization. The well known genetic algorithm approach to optimization is applied and found to be extremely effective.

Much of the effort in this research has been devoted to creating a prototype for testing the proposed approach. As a first step, the sensitivity of computer vision to the environment dictates that the environment be constrained in order to reduce the degree of unpredictability. In this work, the environment is constrained by confining the technique proposed to one vision-based input mode, namely visual tags. Visual tags are shown to greatly reduce the complexity of the vision problem, while at the same time retaining efficacy for certain important applications, including providing assistance in educational activities.

A library of functions has been implemented that detects different types of visual tags. This work contributes to vision technology by providing a generic way of tracking vision-based tangible user interfaces in an environment.

The tag reading algorithms work perfectly on perfect synthetic images; synthetic tags are read correctly on the first try. Problems arise when the images are real. In particular, the color barcodes present significant difficulties for accurate reading. This underscores the importance of the problem of imperfect data in vision-enabled interfaces, especially when, as with the barcode, the intricate details of the image are important. There is little evidence in the literature that this problem has received the attention it requires. For any given new vision-enabled interface, there will be conditions that can easily arise where it can fail completely, mainly due to poor choice of parameters. This dissertation presents one approach that enables the selection of suitable parameters, and hopefully provides ideas for similar approaches in other problems where uncertain inputs pose a challenge. While it may not be possible to produce systems that perform well under all circumstances, the goal of this work is to enable users to automatically obtain the input parameters that can give the more accurate performance in their environment.

7.2 Future Work

We saw in our experiments how handy it is to work with Plexiglas as a non-reflecting tabletop. The children are able to move objects around on it naturally and without getting in the line of sight of the camera (a common problem in vision-based interaction). Yet Plexiglas adds noise that sometimes, in combination with other factors, severely degrades image quality. We observed, for example, that the percentage of correct readings increased significantly when objects were removed from the Plexiglas

and viewed directly. Some future work should focus on resolving this dilemma. Instead of abandoning Plexiglas altogether, the problem should be studied in order to understand and model the noise Plexiglas causes. This will allow it to be accounted for when reading tags. Alternatively the noise model can be used to preprocess images and produce ones that are free of this particular type of noise before reading.

Future work should also continue to earnestly address uncertainty in visual inputs, and others are urged to participate in this important effort in order to ensure robust operation of vision-based systems, without which the technology risks remaining confined in research labs. It is hard for vision-based technology to gain widespread acceptance if the methods produced only work under very strong assumptions that are rarely true in practice. Thinking of the search for suitable input parameters as an optimization problem is one approach that seems to hold great promise. Genetic algorithms provide many possibilities and variations to explore, which is one of the reasons why they are such a popular research tool, not to mention that they have an excellent record of success in many fields.

BIBLIOGRAPHY

1. Ahrens, B. Genetic Algorithm Optimization of Superresolution Parameters *The 2005 Conference on Genetic and Evolutionary Computation*, ACM Press, Washington DC, USA, 2005

2. Alander, J.T., Mantere, T. and Turunen, P., Genetic Algorithms Based Software Testing. In *Proceedings of International Conference on Artificial Neural Nets and Genetic Algorithms*, (1997), Springer-Verlag.

3. Aoki, H. and Matsushita, S., Balloon Tag: (In)visible Marker Which Tells Who's Who. In *Proceedings of Fourth International Symposium on Wearable Computers*, (Atlanta, Georgia, 2000), IEEE Computer Society, 181-182.

4. Ashley, T. Meosis, meosis.jpg, 702 x 966, 52.4 KB, http://www.med.yale.edu/genetics/ashley/text/pictures/meiosis.JPG.

5. Beardsley, P. Detecting Visual Tags, 2001, http://www.merl.com/projects/visual-tags/.

6. Beizer, B. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

7. Blamire, J. Science at a Distance: Biological Information, 2005, http://www.brooklyn.cuny.edu/bc/ahp/SDV2.html.

8. Bowyer, K.W. and Phillips, P.J. Overview of Work in Empirical Evaluation of Computer Vision Algorithms. In Bowyer, K.W. and Phillips, P.J. eds. *Empirical Evaluation Techniques in Computer Vision*, IEEE Computer Society, 1998.

9. Bruckman, A. and Bandlow, A. HCI For Kids. In Jacko, J.A. and Sears, A. eds. *The Human-Computer Interaction Handbook*, Lawrence Erlbaum Assoc, Mahwah, NJ, 2003.

10. Bushnell, M.E. and Jones, M.S., Get a Grip: Managing RSI at MIT. In *Proceedings of 22nd Annual ACM SIGUCCS Conference on User Services*, (Yipsilanti, Michigan, 1994), ACM Press, 267-270.

11. Cardani, D. Adventures in HSV Space, 2001, http://www.buena.com/articles/hsvspace.pdf.

12. Cinque, L. and Cucchiara, R., Optimal Range Segmentation Parameters Through Genetic Algorithms. In *Proceedings of International Conference on Pattern Recoginition*, (2000), IEEE Computer Society.

13. Cowan, C., Wagle, P., Pu, C., Beattie, S. and Walpole, J., Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of*

DARPA Information Survivability Conf. And Exp, (2000), IEEE Computer Society.

14. Czaja, S.J. and Lee, C.C. Designing Computer Systems for Older Adults. In Jacko, A.J. and Sears, A. eds. *The Human-Computer Interaction Handbook*, Lawrence Erlbaum Assoc, Mahwah, NJ, 2003.

15. Davis, L. *Handbook of Genetic Algorithms*. International Thompson Computer Press, Boston, MA, 1996.

16. de Ipina, D.L., Mendonça, P. and Hopper, A. TRIP: A Low-Cost Vision-Based Location System for Ubiquitous Computing. *Personal and Ubiquitous Computing Journal*, *6*. 206-219, 2003.

17. De Silva, G.C., Lyons, M.J., Kawato, S. and Tetsutani, N., Human Factors Evaluation of a Vision-Based Facial Gesture Interface. In *Proceedings of Computer Vision and Pattern Recognition Workshop*, (2003), IEEE Computer Society.

18. Dolan, A. GA Playground: Java Genetic Algorithms Toolkit, 1999, http://www.aridolan.com/ga/gaa/gaa.html.

19. Doong, R.-K. and Frankl, P.G. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Transactions on Software Engineering Methodology*, 1994.

20. Dougherty, S. and Bowyer, K. Objective Evaluation of Edge Detectors Using a Formally Defined Framework. In Bowyer, K.W. and Phillips, P.J. eds. *Empirical Evaluation of Computer Vision Algorithms*, IEEE Computer Society, 1998.

21. Goldberg, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

22. Harmanani, H. and Karablieh, B., A Hybrid Distributed Test Generation Method Using Deterministic and Genetic Algorithms. In *Proceedings of 9th International Database Engineering & Application Symposium*, (2005), IEEE Computer Society.

23. Haupt, R.L. and Haupt, S.E. *Practical Genetic Algorithms*. John Wiley & Sons, New York, NY, 1998.

24. Holland, J.H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

25. Hsiao, M.S., Rudnick, E.M. and Patel, J.H., Automatic Test Generation Using Genetically-Engineered Distinguishing Sequences. In *Proceedings of 14th IEEE VLSI Test Symposium*, (1996), IEEE Computer Society, 216-223.

26. Huang, C.-J., Not Just Intuitive: Examining the Basic Manipulation of Tangible User Interfaces. In *Proceedings of CHI '04*, (Vienna, Austria, 2004), ACM Press, 1387-1390.

27. Huang, T.S. and Pavlović, V.I., Hand Gesture Modeling, Analysis, and Synthesis. In *Proceedings of The International Workshop on Automatic Face- and Gesture-Recognition*, (Zurich, Switzerland, 1995), IEEE Computer Society, 73-79.

28. Iannizzotto, G., Costanzo, C., Lanzafame, P. and Rosa, F.L., Badge3D for Visually Impaired. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, (2005), IEEE Computer Society.

29. Ishii, H. and Ullmer, B., Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In *Proceedings of CHI*, (Atlanta, GA, 1997), ACM Press.

30. Jorgensen, P. *Software Testing: A Craftman's Approach*. CRC Press, 2002.

31. Kamberova, G. and Bajcsy, R. Sensor Errors and the Uncertainties in Stereo Reconstruction. In Bowyer, K.W. and Phillips, P.J. eds. *Empirical Evaluation Techniques in Computer Vision*, IEEE Computer Society, 1998.

32. Kim, H. and Fellner, D.W., Interaction with Hand Gesture for a Back-Projection Wall. In *Proceedings of Computer Graphics International*, (2004), IEEE Computer Society.

33. Lee, G. and Bulitko, V. GAMM: Genetic Algorithms with Meta-Models for Vision *2005 Conference on Genetic and Evolutionary Computation*, ACM Press, Washington DC, USA, 2005

34. Liu, X., Kanungo, T. and Haralick, R.M., Statistical Validation of Computer Vision Software. In *Proceedings of DARPA Image Understanding Workshop*, (Palm Springs, CA, 1996), 1533-1540.

35. Ljungstrand, P., Redstrom, J. and Holmquist, L.E., Webstickers: Using Physical Tokens to Access, Manage and Share Bookmarks on the Web. In *Proceedings of ACM Conference on Designing Augmented Reality Environments*, (Elsinore, Denmark, 2000), ACM Press, 23-31.

36. Macromedia. Macromedia Open Architecture, http://www.macromedia.com/support/xtras/info/moa.html.

37. Magee, J.J., Scott, M.R., Weber, B.N. and Betke, M., EyeKeys: A Real-Time Vision Interface Based on Gaze Detection from a Low-Grade Video Camera. In *Proceedings of IEEE Workshop on Real-Time Vision for Human-Computer Interaction*, (Washington, D.C., 2004), IEEE Computer Society, 159-166.

38. Majaranta, P. and Räihä, K.-J., Twenty Years of Eye Typing. In *Proceedings of The Symposium on Eye Tracking Research and Applications*, (New Orleans, LA, 2002), ACM Press, 15-22.

39. Matovic, K., Psik, T. and Wagner, I. Tangible Image Query. *Lecture Notes in Computer Science*, *3031*. 31-42, 2004.

40. Mbogho, A.J.W. and Scarlatos, L.L., Towards Reliable Computer Vision-Based Tangible User Interfaces. In *Proceedings of IASTED International Conference on Human-Computer Interaction*, (Phoenix, AZ, 2005), ACTA Press, 155-160.

41. Mbogho, A.J.W., Scarlatos, L.L. and Jaworska, M., Teaching with Tangibles: A Tool for Dichotomous Sorting Activities. In *Proceedings of CHI-SA 2006 (to appear)*, (Cape Town, South Africa, 2006), ACM Press.

42. Michael, C. and McGraw, G., Automated Software Test Data Generation for Complex Programs. In *Proceedings of 13th IEEE International Conferance on Automated Software Engineering*, (1998), IEEE Computer Society, 136-146.

43. Michael, C., McGraw, G.E., Schatz, M.A. and Walton, C.C., Genetic Algorithms for Dynamic Test Data Generation. In *Proceedings of Automated Software Engineering Conference*, (1997), IEEE Computer Society, 3-5.

44. Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York, NY, 1996.

45. Moran, T.P., Saund, E., van Melle, W., Gujar, A.U., Fishkin, K.P. and Harrison, B.L., Design and Technology for Collaborage: Collaborative Collages of Information on Physical Walls. In *Proceedings of UIST '99*, (Asheville, NC, 1999), ACM Press, 197-106.

46. Nguyen, H.Q. *Testing Applications on the Web*. John Wiley & Sons, 2003.

47. NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002, http://www.nist.gov/director/prog-ofc/report02-3.pdf.

48. Pargas, R.P., Harrold, M.J. and Peck, R.R. Test-Data Generation Using Genetic Algorithms. *Journal of Software Testing, Verification, and Reliability*, *l.9*, 1999.

49. Parker, J.R. *Practical Computer Vision Using C*. John Wiley & Sons, New York, NY, 1994.

50. Pavlidis, T. *Algorithms for Graphics and Image Processing*. Computer Science Press, Rockville, MD, 1982.

51. Peixoto, P. and Carreira, J., A Natural Hand Gesture Human Computer Interface Using Contour Signatures. In *Proceedings of IASTED International Conference on Human-Computer Interaction*, (Phoenix, AZ, 2005), ACTA Press, 19-24.

52. Pfeiffer, P. and Heintzelman, M., Machines, Statues and People: Strategies for Promoting RSI Awareness in Computing Curricula. In *Proceedings of 28th SIGCSE Technical Symposium on Computer Science Education*, (San Jose, California, 1997), ACM Press.

53. Quek, F. Eyes in the Interface. *Image and Vision Computing*, *13* (6). 511-525, 1995.

54. Quek, F., Non-Verbal Vision-Based Interfaces. In *Proceedings of International Workshop in Human Computer Interface Technology*, (Aizuwakamatsu, Fukushima, Japan, 1995).

55. Quek, F., Mysliwiec, T. and Zhao, M., FingerMouse: A Freehand Pointing Interface. In *Proceedings of International Workshop on Automatic Face and Gesture Recognition*, (Zurich, Switzerland, 1995), 372-377.

56. Raskar, R., Beardsley, P., van Baar, J., Wang, Y., Dietz, P., Lee, J., Leigh, D. and Willwacher, T. RFIG Lamps: Interacting with a Self-Describing World via Photosensing Wireless Tags and Projectors. *ACM Transactions on Graphics*, *23* (3), 2004.

57. Rätzmann, M. and Young, C.D. Galileo Computing Software Testing and Internationalizaion, Lemoine International Inc., 2003, http://www.lisa.org/interact/2003/SoftwareTesting.zip.

58. Rekimoto, J. and Ayatsuka, Y., CyberCode: Designing Augmented Reality Environments with Visual Tags. In *Proceedings of ACM Conference on Designing Augmented Reality Environments*, (2000), ACM Press, 1-10.

59. Rohs, M., Visual Code Widgets for Marker-Based Interaction. In *Proceedings of International Conference on Distributed Computing Systems Workshops*, (Columbus, OH, 2005), IEEE Computer Society, 506-513.

60. Ruskey, F. Information on Pentomino Puzzles, http://www.theory.cs.uvic.ca/~cos/inf/misc/PentInfo.html.

61. Scarlatos, L.L. TICLE: Using Multimedia Multimodal Guidance to Enhance Learning. *Information Sciences*, *140*. 85-103, 2002.

62. Scarlatos, L.L. and Landy, S.S., Experiments in Using Tangible Interfaces to Enhance Collaborative Learning Experiences. In *Proceedings of CHI '01*, (Seattle, WA, 2001), 257-258.

63. Scharstein, D. and Szeliski, R., A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. In *Proceedings of IEEE Workshop on Stereo and Multi-Baseline Vision*, (2001).

64. Schwarz, M.W., Cowan, W.B. and Beatty, J.C. An Experimental Comparison of RGB, YIQ, LAB, HSV, and Opponent Color Models. *ACM Transactions on Graphics*, *6* (2). 123-158, 1987.

65. Scott, D., Sharp, R., Madhvapeddy, A. and Upton, E. Using Tags to Bypass Bluetooth Device Discovery. *ACM Mobile Computing and Communications Review*, *9* (1). 41-53, 2005.

66. Semacode. Semacode, http://semacode.org/about/technical/.

67. Skarbek, W. and Koschan, A. Colour Image Segmentation - A Survey *Technical Report 94-32*, Technical University of Berlin, Dept of Computer Science, 1994, http://.imaging.utk.edu/~koschan/paper/coseg.pdf.

68. Smith, A.R., Color Gamut Transform Pairs. In *Proceedings of 5th Annual Conference on Computer Graphics and Interactive Techniques*, (ACM Press, 1978), 12-19.

69. Stafford-Fraser, Q. and Robinson, P., BrightBoard: A Video-Augmented Environment. In *Proceedings of SIGCHI conference on Human factors in computing systems: common ground*, (Vancouver, British Columbia, Canada 1996), ACM Press, 134-141.

70. STORM. Software Testing Online Resources at Middle Tennessee State University, http://www.mtsu.edu/~storm/.

71. Sturman, D.J. and Zeltzer, D. A Survey of Glove-Based Input. *IEEE Computer Graphics and Applications*, *14*. 30-39, 1994.

72. Toye, E., Sharp, R., Madhvapeddy, A. and Scott, D. Using Smart Phones to Access Site-Specific Services. *Pervasive Computing*, *1536-1268*. 60-66, 2005.

73. Tracey, N., Clark, J., Mander, K. and McDermid, J. Automated Test-Data Generation for Exception Conditions. *Software—Practice and Experience*, 2000.

74. Tracey, N., Clark, J., McDermid, J. and Mander, K., Integrating Safety Analysis with Automatic Test-Data Generation for Software Safety Verification. In *Proceedings of 17th International System Safety Conference*, (1999).

75. Trucco, E. and Verri, A. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, 1998.

76. Turk, M. and Robertson, G. Perceptual User Interfaces. *Communications of the ACM*, *43* (3). 33-34, 2000.

77. Underkoffler, J. and Ishii, H., Illuminating Light: An Optical Design Tool with a Luminous-Tangible Interface. In *Proceedings of CHI '98*, (1998), ACM Press.

78. Wang, J. Hybrid Genetic Algorithms for Reliability Assessment of Structural System, Engineering Department, City University of New York, New York, Ph.D., 1994.

79. Ye, G., Corso, J.J. and Hager, G.D., Gesture Recognition Using 3D Appearance and Motion Features. In *Proceedings of Computer Vision and Pattern Recognition Workshops*, (2004), IEEE Computer Society.

80. Zhang, H., Zhou, X., Sun, J. and Zhang, J. A Novel Medical Image Registration Method Based on Mutual Information and Genetic Algorithm *International Conference on Computer Graphics, Imaging and Visualization (CGIV'05)*, IEEE Computer Society, 2005