Examensarbete
LITH-ITN-MT-EX--07/045--SE

ImBrowse.NET - A Software System for Image Database Search

Joel Erving Bjelkholm

2007-09-17

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköpings universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LITH-ITN-MT-EX--07/045--SE

ImBrowse.NET - A Software System for Image Database Search

Examensarbete utfört i medieteknik
vid Linköpings Tekniska Högskola, Campus Norrköping

Joel Erving Bjelkholm

Handledare Reiner Lenz
Examinator Reiner Lenz

Norrköping 2007-09-17

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2007-09-17

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _____

Rapporttyp
Report category

Examensarbete
☐ B-uppsats
☐ C-uppsats
☒ D-uppsats

☐ _____

ISBN

ISRN **LITH-ITN-MT-EX--07/045--SE**

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version

Titel
Title

ImBrowse.NET - A Software System for Image Database Search

Författare
Author

Joel Erving Bjelkholm

Sammanfattning
Abstract

Content-based image retrieval systems rely on computer vision to find images based on visual content rather than predefined keywords. Using images as input as opposed to text can provide the user with a novel and in some senses more intuitive way of searching for images. ImBrowse.NET is a platform for developing content-based image retrieval methods. It takes the form of an extendible stand-alone application, written using the Microsoft .NET 3.0 Framework. The application employs principal components analysis to reduce the complexity of the datasets. This report focuses mainly on the architecture of the application with the intention of providing a user with knowledge needed to extend it with new query modes.

Nyckelord
Keyword

Content-Based Image Retrieval, Image Databases, Principal Components

Abstract

Content-based image retrieval systems rely on computer vision to find images based on visual content rather than predefined keywords. Using images as input as opposed to text can provide the user with a novel and in some senses more intuitive way of searching for images. ImBrowse.NET is a platform for developing content-based image retrieval methods. It takes the form of an extendible stand-alone application, written using the Microsoft .NET 3.0 Framework. The application employs principal components analysis to reduce the complexity of the datasets. This report focuses mainly on the architecture of the application with the intention of providing a user with knowledge needed to extend it with new query modes.

Keywords: Content-Based Image Retrieval, Image Databases, Principal Components Analysis

Acknowledgements

My supervisor and examiner Reiner Lenz for being there every step of the way. My brother Terje Erwing Bjelkholm for your expertise with .NET and C#. Gaia System AB for graciously allowing me to use their facilities even though I no longer worked there. Thank you.

Contents

Chapter 1: Introduction 2
1.1 Purpose 2
1.2 Method 2
1.3 Limitations 2
1.4 Report structure 2
Chapter 2: Initial Concept 3
2.1 Google Desktop 3
2.2 Why Google Desktop was a bad idea 3
Chapter 3: The .NET Framework 5
3.1 Windows Presentation Foundation 5
Chapter 4: Content-based Image Retrieval 7
4.1 User Query 7
4.2 Similarity Measures 8
4.3 CBIR vs. Textual search 8
Chapter 5: Principal Components Analysis 9
5.1 Computing the transform 9
Chapter 6: Feature Extractors 11
6.1 RGB-histogram 11
Chapter 7:Application Overview 12
7.1 Class Description 12
Chapter 8:Implementing a new feature extractor 15
8.1 Start-up 15
8.2 Building the database 15
8.3 Updating the database 15
8.4 Adding and removing images 16
8.5 Updating the correlation and transform matrices 16
8.6 Querying the database 17
Chapter 9: User Manual 18
9.1 Step-by-step Guide 20
Chapter 10: Interface 22
10.1 The Main Window 22
10.2 Making a query 22
10.3 Viewing the result set 25
10.4 The Photo Viewer 25
Chapter 11: Conclusions 27
11.1 Future work 27
Appendix A: Creating your first feature extractor A1

Chapter 1
Introduction

Searching for images based on visual content rather than keywords can be more intuitive and in many ways more useful. One of the major problems in using image processing and computer vision to search for images is that it's very computationally complex and thus requires considerably more time than text based searches. This can result in both in unrealistic query times and very large amounts of metadata, especially for large databases. In order to make such searches practical, a method for preprocessing the images and reducing the complexity of the extracted data must be implemented. This will allow for fast queries and a relatively small amount of excess data to store per image.

1.1 Purpose

The purpose was to develop a platform for easy implementation of new content-based image retrieval methods. It should provide an efficient database, a graphical user interface and a comparatively fast query speed. It should also be easy to extend, requiring a developer to be exposed to only a very small portion of the source code.

1.2 Method

The method used in this report extracts image features and stores them in high dimensional vectors. A technique known as Principal Components Analysis is then used to project the vectors on a lower dimension vector space. This has both the benefit of saving disk space and reducing the query time.

1.3 Limitations

The application is designed to be able to handle databases of up to 100,000 images. This is however not a hard limit in any way. Since the application is written using the .NET Framework, it will as of now only work on systems running Microsoft Windows.

1.4 Report structure

First the initial concept for the application is discussed, followed by an overview of some of the technologies and techniques used. Next the application itself, its source code and interface are detailed. Finally conclusions and future work are presented.

Chapter 2
Initial Concept

The initial concept was to implement a desktop version of a feature based search tool for images. The purpose of this application was to act as a platform for further development of feature based search methods. This was to be an alternative method to developing this type of methods using The MathWorks MATLAB.

One of the most important aspects of this platform was to be simplicity for the user. The user would only have to write the code for the actual extraction of features from an image, everything else was to be automated.

2.1 Google Desktop

In the early stages of development, the option of writing the application as a Google Desktop (GD) plug-in was explored. GD is a search application developed by Google that allows the user to search for files on the local machine in a way similar to searching for web pages using Google's online search engine, [1]. GD builds up a database using a file crawler. When the crawler finds a file with a file extension registered by one of its components, the component parses information from the file and stores it in the database. How this is done and what information to be stored is up to the component.

Some of the interesting aspects of this were that GD provides a file crawler and a basic user interface. This would allow more time to be spent on developing the other parts of the application.

2.2 Why Google Desktop was a bad idea

After careful consideration and experimentation, the GD approach was deemed unsuccessful. One of the main reasons for this was the problems associated with developing GD plug-ins using .NET and C#. GD is written using COM (Windows Component Object Model) and C++ and although there exists third party C# wrappers, these are not supported or even approved by Google. In earlier versions of the GD API, simple C# examples were included, but with each iteration Google seemed to be moving farther and farther away from the .NET approach.

However, this was not the only problem with GD. One of its main attractions, the crawler, also had its limitations. The GD database is built up file by file, with the crawler working while the system is idle. In order to perform the compression of the extracted feature information needed for fast and memory efficient image searches, all the image files on the

system needs to be examined first. This meant that the uncompressed feature data would have to be saved on disk until all images had been processed before performing the compression. This could potentially lead to very much wasted hard drive space.

The last and perhaps most problematic thing about GD is how it stores the parsed information. A set of predefined XML-schemas are used and these cannot be altered or expanded by a developer. This rigidness in addition to the fact that the bulk of information extracted from the images would more efficiently be stored in a binary format, rather than as XML data, led to the conclusion that the advantages to develop the application as a GD plug-in were slim at best, since neither the crawler nor XML-schemas were ideal.

Chapter 3
The .NET Framework

The .NET Framework is Microsoft's attempt to facilitate the development of applications. It does this mainly in two ways: it provides a vast library of predefined classes and it standardizes the way programming languages interact with the framework. This means that code written in any of the .NET compliant languages is not only usable, but extendable in other .NET applications, independent of the language they are written in. This is achieved by a type definition that is shared among the languages (i.e. a C# string is the same as a Visual Basic.NET string). Developers benefit greatly from this as migrating from one .NET language to another only requires learning the basic syntax of that language. It also allows different parts of an application to be written using different languages.

Another feature of .NET that can help reduce the complexity of code is the option to let .NET handle the memory management. This is referred to as *managed code* and lets the developer focus on other areas. Unlike JAVA, which also uses managed code, a .NET application can be written completely or in part in unmanaged, so called *un-safe code*. This is particularly useful in code segments where performance is crucial. Although managed code is less efficient than unmanaged code in terms of processor cycles, it is generally less prone to bugs and lets the developer focus on the other aspects of programming. It can be argued that this produces applications that are comparable or even superior in efficiency to those written in an unmanaged language in a real life scenario, where development time is limited.

.NET code is also platform independent, at least in theory. It borrows the idea of intermediate code from JAVA. Code is first compiled to *Common Intermediate Code* (CIL) and then further compiled by the *Common Language Runtime* (CLR) to platform specific, machine-readable code (see fig. 3.1). Unlike the JAVA Virtual Machine, the code is compiled into an executable application, rather than interpreted. However, as of yet .NET is only available for Microsoft Windows. There has been talk of bringing .NET to both Mac OS and Unix-based systems such as Linux. Although there exists some excellent open-source projects that attempt to do just that, .NET cannot be considered truly platform independent as long as there is no official, Microsoft supported solution.

3.1 Windows Presentation Foundation

Windows Presentation Foundation (WPF) is a graphical subsystem of the .NET Framework. It is similar to *Windows Forms* but more flexible and powerful. WPF is closely linked to *Extensible Application Markup Language* (XAML), a declarative XML-based language

developed by Microsoft. WPF attempts to separate the business logic of an application from its Graphical User Interface (GUI). The GUI can be created by a developer without any knowledge of the inner workings of the classes and methods of a program. This works in a way similar to how the content of a web page, written in HTML, is separate from its presentation, written in CSS. As WPF can be written using XAML, it can be scripted rather than coded, minimizing the requirement for the GUI developer to have extensive programming skills. WPF was developed primarily for Windows Vista. Like in Vista, all graphic elements in WPF are vector based and scalable. This means that the GUI developer does not need to take into account the fact that different systems have different resolutions.

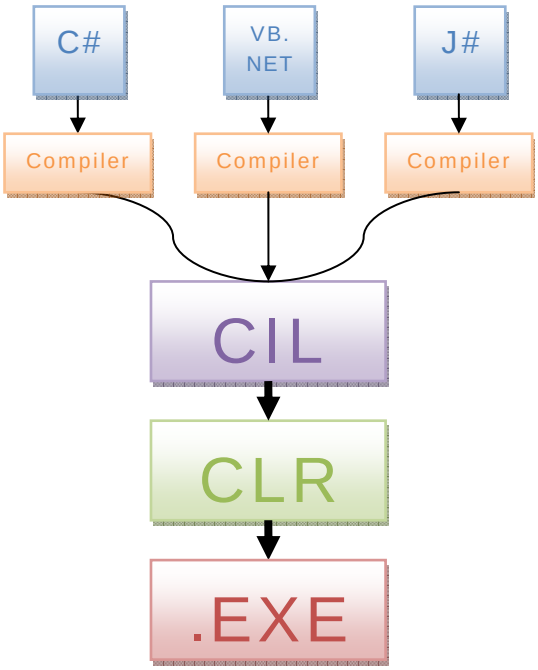

Figure 3.1

Chapter 4
Content-based Image Retrieval

Content-based Image Retrieval (CBIR) is the application of computer vision techniques to the problem of finding an image in a large image database. This approach is fundamentally different to a metadata based search, which is generally dependant on all the images in the database being preprocessed and tagged by a human user.

CBIR works by analyzing specific characteristics of images and then comparing them with the characteristics of the query. Such characteristics are referred to as *features* and consist of colors, textures and shapes. Color is the most widely used feature and is commonly described using a *histogram*. Textures are patterns, such as a periodic repetition of an element (like checkers, stripes or dots), or areas of similarity or slow change in color. Shapes are easily understood, but hard to define. They are generally described either by their border outline, so called *boundary-based features* or by their entire shape, *region-based features*. Specific systems can also implement their own high-level features. These are more specific and might include things like human faces in a surveillance system. [2]

4.1 User Query

Querying a CBIR system is very different from performing a search in a text based system. The most intuitive query method lets the user supply an image for which the system attempts to find close matches. This is referred to as *Query by Example* (QBE). The query image can either be a pre-existing image, like a digital photograph, or created by the user using some basic type of drawing software. Both of these methods have their problems however. Supplying a pre-existing image requires the user to already have found an image similar to the one he is looking for. Creating an image requires both a certain degree of artistic prowess and might be much too time-consuming.

Another, less intuitive, but more exact approach is *Query by Feature* (QBF). This lets the user define feature properties, like “at least 20% of all the pixels in the image should have a red value of more than 150”. In addition to being more precise, a QBF system also sidesteps the potentially frustrating problem of requiring the user to supply a query image. Even so, this type of query is reserved only for the expert user and even then might only be more useful than a QBE system in specific cases. [2]

4.2 Similarity Measures

Once a query has been made, the system must somehow decide the similarity between the query image and the images in the database. Constructing an accurate and efficient similarity measure is one of the main challenges in designing a CBIR system. Such a measure should ideally incorporate the following basic properties:

Perceptual Similarity: Image similarity is generally measured in feature space, where the greater the distance, the more dissimilar the images are. Images that are close to each other should appear more similar to a human viewer than those lying farther apart.

Efficiency: A CBIR system may contain thousands or even millions of images. It is therefore paramount that the measure is computed rapidly for each image.

Scalability: The system should be able to perform well even for very large databases. The time factor for a naïve implementation scales linearly with the number of images in the database (*O(N)*), but it has been shown that this can be reduced to *O(log N)* under certain conditions by using multi-dimensional indexing techniques, such as quad-trees.

Metric: The measure should be metric, since the following properties seem natural requirements.

- Constancy of self-similarity: The distance between an image to itself should preferably be zero, or at least equal to a constant, independent of the image.
- Minimality: An image should be more similar to itself than to any other image.
- Symmetry: If image A is similar to image B, image B must also be similar to image A.
- Transitivity: If image A is very similar to image B and Image B is very similar to image C, then image A must be similar to image C. This might not hold true if the sequence of images is long enough, i.e. image A might be very different from image Z, even though all intermediate images may be very similar to their neighbors.

Robustness: The system should be able to handle changes in the imaging conditions of the database images. [2]

4.3 CBIR vs. Textual search

Textual search is intuitive to most users and is both simple and fast to process, but is coupled with several major problems when applied to image retrieval. Such searches are dependent on metadata such as captions and keywords which are both laborious to produce and subjective. When it comes to systems that search for images on the Internet, providing proper keywords becomes almost impossible, even when coupled with automated text-parsers.

CBIR can provide an intuitive method for the general user to search for images as well as allowing the expert user use advanced techniques to find images with certain characteristics. The main problem with CBIR systems for the user is providing a query image. One way of solving this is by using a textual search to find an initial image and then use that image as a query for the CBIR system. CBIR systems are very useful when indexing very large image databases, such as the collection of images on the Internet, as the process is completely automated.

Chapter 5
Principal Components Analysis

Principal components analysis (PCA) is a method used in statistics to reduce the size and complexity of a dataset. This is done by projecting a high dimensional vector space on one with a lower dimension. This transform is done in such a way that the component with the highest variance will lay on the first coordinate, the second highest on the second coordinate and so on in descending order. This technique is also known as the discrete Karhunen-Loève transform, the Hotelling transform or proper orthogonal decomposition, depending on the field of application. It has been proved that this is the optimal way to map N-dimensional vectors on lower K-dimensional vectors, in the sense that it minimizes the Euclidian distance between the original vector and the mapped vector. [3]

5.1 Computing the transform

The goal is to construct a transform T that maps the K-dimensional row-vectors of a data matrix X on the L-dimensional column-vectors of a matrix Y, where K>>L.

$$Y = T\{X\}$$

Given a data matrix X of size N*K the *correlation matrix* C is first computed by multiplying the transpose of X with itself. C is a square matrix of size K*K.

$$C = X^T * X$$

Next, the matrix of eigenvectors V that diagonalizes the correlation matrix C is computed:

$$V^{-1}CV = D$$

where D is the diagonal matrix of eigenvalues of C. Both V and D are of size M*M where M is the number of eigenvectors of C. Each column in V represents one eigenvector and is matched by an eigenvalue in the corresponding column on the diagonal of D. V and D are sorted column-wise in decreasing eigenvalue order, making sure to contain the correct pairings of the eigenvectors and eigenvalues. A sub-matrix of V called the transformation matrix W is created by using the eigenvectors corresponding to the L highest eigenvalues, i.e. the L first columns of V. W is of size L*K.

$W[p,q] = V[p,q]$ for $p = 1...M \quad q = 1...L$

The last step consists of computing the matrix Y by multiplying the transform matrix T with the transpose of the data matrix X:

$Y = W * X^T$

Y is of size L*N and is the best L-dimensional projection of the K-dimensional vectors in the original data matrix X.

Chapter 6
Feature Extractors

A *feature extractor* is in the context of this report defined as a function that computes a set of characteristics from a digital image and stores the results as a high-dimensional vector. For a more detailed discussion on such characteristics, see chapter 4.

ImBrowse.NET was designed to facilitate the development of feature extractors. In this sense, it is more a development platform than an application. For demonstration purposes however, it comes with a single feature extractor already implemented – the RGB-histogram.

6.1 RGB-histogram

In image analysis, a *histogram* is a graphical representation of the probability distribution of the pixel values of the image. In its simplest form, the histogram represents the distribution of pixel intensity or lightness (i.e. the grayscale value of the pixels). A histogram is discreet and the possible values of the pixel are generally divided into a number of ranges called *bins*. The higher the number of bins is the higher the accuracy of the histogram but also the larger its size.

An RGB-histogram is a three dimensional histogram that shows the distribution of pixels having a specific RGB-value. It is not as easily visualized as a simple intensity histogram, but the concept is simple. The probability for any given RGB-value lies in a cube matrix with width, height and depth representing red, green and blue. Since the size of the histogram is the number of bins per channel cubed, it is important to keep the bins rather large. The RGB-histogram implemented in this application has eight bins for each channel with a total of 512 bins. After PCA compression, 20 eigenvectors are used to store the histograms.

Chapter 7
Application Overview

The source code for the application is divided into a number of files. Each file contains a logical grouping of classes and methods. The files can be further grouped into either having to do with the business logic or the user interface. A short overview of the files:

- **Business logic**
  - Database.cs – Contains classes that handle data storage and query.
  - Mapack.dll – Contains classes that allow fast matrix operations.
  - FeatureExtractors.cs – Contains methods that handle the feature extraction.
- **User interface**
  - MainWindow.xaml – Contains mark-up code for the main window.
  - MainWindow.xaml.cs – Code-behind for the main window.
  - PhotoView.xaml – Contains mark-up code for the single photo window.
  - PhotoView.xaml.cs – Code-behind for the single photo window.
  - Data.cs – Contains a class that describes a single image and its metadata.
  - Conterters.cs – Contains classes that convert image header data into human-readable format.
  - App.xaml – Contains mark-up code that instantiates objects needed for the main window.
  - App.xaml.cs – Code-behind for instantiating the main window.

A more detailed description of the business logic classes and their methods follows in the next section.

7.1 Class Description

- **Database.cs**
  - Class: ParentDirectory
    This class stores information on the location of a directory. It is used when adding a folder of images to the database, both to store the location of that folder and whether or not to include its sub-folders in the database.
  - Class: FilePath
    This class stores information on the location and status of a single image file. When adding a folder to the database, an instance of this class is created for every image.

Flags that check whether the image has been removed or modified since the last time the application was run are also stored.

- Class: Database
  This is the main database class and it contains all of the methods associated with creating, updating and querying the database.
  - Method: GetFilePath
    This method builds a list of all the image-files in the directories specified in its in-parameter.
  - Method: GetDirectories
    A recursive method that builds a list of all the image-files in the directories specified in its in-parameter that have a flag indicating whether or not to include its sub-directories set to true.
  - Method: BuildPathPosDictionary
    This method builds a Dictionary with the paths of the files in the database as keys and their respective position in the database.
  - Method: UpdateDatabase
    This method updates the database with new images and removes deleted or moved image files.
  - Method: BuildDatabase
    This method completely rebuilds the database. The first time the application is started, this method will be run. Depending on the number of images in the database as well as the number of implemented feature extractors and their complexity, this might be very time consuming.
  - Method: BuildDatabase [Single feature extractor]
    This method adds the information extracted from the images in the database by a single feature extractor. This is useful when implanting a new feature extractor, as it doesn't require the entire database to be rebuilt.
  - Method: BuildFeatureMatrix
    This method constructs a matrix containing the uncompressed feature vectors of all images in the database.
  - Method: CheckTransformMatricesAccuracy
    This method checks to see whether the transform matrices need to be recalculated. This may happen when a large number of images with very different characteristics than those used to originally build the database are added.
  - Method: QueryDatabase
    The method used to query the database. It takes an image and returns a list of the best matches found in the database, for the chosen feature.
  - Method: SerializeFilePaths
    This method saves a list of the paths to the images in the database in an XML-file for later reference.
  - Method: DeserializeFilePaths
    This method creates objects of the information of the paths to the images stored in an XML-file.
  - Method: FindBestMatches
    This method finds the best matches to a query by sorting a list of results.
  - Method: LoadAppSettings
    This method loads the user settings defined in the application's configuration file.
  - Method: UpdateAppSettings
    This method updates the user settings in the application's configuration file.

- Method: WriteBinaryMatrixData
  This method stores the information extracted from the images by a single feature extractor in a binary format on disk.
- Method: ReadImageMatrixData
  This method loads all information extracted from the images by a single feature extractor to memory.
- Method: ReadImageMatrixData
  This method loads chosen parts of the information extracted from the images by a single feature extractor to memory.

- **FeatureExtractors.cs**
  - Class: FeatureExtractor
    This static class contains the code for all implemented feature extractors. The feature extractors are methods and a switch is used to choose feature extractor.
    - Method: ExtractFeatures
      This method acts as a switch, controlling which feature extractor is called.
    - Method: GetPixelData
      This method extracts BGR32 pixel data from an image. It is called from feature extractor to get access to the pixel data. The BGR32 format is a sRGB format with 32 bits per pixel. Each color channel blue, green, and red) is allocated 8 bits per pixel. Note the reversed order or the color channels. An overload of this method also provides the width and height of the image in pixels.
    - Method: CalculateRGBHistogram
      This method extracts an uncompressed RGB histogram from an image. This is the only feature extractor implemented in the application in its current state and is designed as an example on how they can be written.
- **Mapack.dll**
  Mapack is a .NET class library for basic linear algebra computations. It supports Norm1, Norm2, Frobenius Norm, Determinant, Infinity Norm, Rank, Condition, Trace, Cholesky, LU, QR, single value decomposition, least squares solver and eigenproblems. It is written by Lutz Roeder and used with permission in this application. For further information, please visit Lutz Roeder's webpage: *http://www.aisto.com/roeder/dotnet*

Chapter 8
User Manual

8.1 Start-up

On start-up a number of methods to keep the database up to date are run. Depending on the user settings, these methods can be suppressed in order to decrease the start up time.

8.2 Building the database

Before the application can be used, a database of searchable images needs to be built. The user can add folders that he wishes to be searchable by adding their paths to the applications configuration file. The full path needs to be added to comma separated list of the *ParentDirectory* key. If the subfolders should also be added, add *true* to the comma separated list of the *IncludeSubDirectories* key; otherwise add *false* to the key. Once all folders have been chosen, save and close the configuration file and run the application. This will start the process of building the database. Depending on the number of images and which feature extractors are used, this can take up to several hours. It is highly recommended that all folders containing images the user wants to be searchable are added before the database is built. Adding images later will affect the quality of the search.

8.3 Updating the database

A method for updating the database without the need to rebuild it exists. Unless the user has disabled this feature, this is always run on start up. All image files in the folders specified by the user are checked. If any new images are found or if any images already in the database have been modified since the last database update, their features are extracted with the appropriate feature extractors and added to the database.

Since the search for matches to a query image is performed using a transform matrix calculated from the uncompressed feature matrix, this matrix cannot be updated without rebuilding the database. This may lead to diminishing quality of the search results if too many images are added to the database using only the update process. This is especially apparent if the content of the added images differs greatly from the images used to build the database. To avoid this, the quality of the transform matrix is always checked after updating the database. If the quality drops below a predefined threshold value, the user is alerted and recommended to rebuild the database. Since each feature extractor has its own transform matrix, it is likely that not the entire database needs to be rebuilt.

8.4 Adding and removing images

The update method works by first reading a list of all the files that were in the database after the last update and then creating a list of all the files that are now in the searchable directories. These lists are then compared for any discrepancies. If a new file is found its features are extracted and added to the database. If a file has been modified since the last database update, its features are recalculated. If a file is found to have been removed, it is also removed from the database.

8.5 Updating the correlation and transform matrices

Even though the update method keeps the information in the database current, the correlation matrices and their corresponding transform matrices are not so easily handled. Since the correlation matrix for a given feature is computed using the uncompressed feature data, the only way to update it without any risk of accuracy loss would be to rebuild the database. Since this is extremely time consuming, another less accurate but much more efficient approach is used.

A new correlation matrix is calculated by adding the original matrix weighted with the number of images it was calculated from to a correlation matrix of the added images weighted with the number of new images. This is then divided by the total number of images. If C_N is the original correlation matrix, C_K is the added correlation matrix, N is the number of images used for C_N and K the number for C_K, then the new correlation matrix C_{N+K} is:

$$C_{N+K} = \frac{\left(N * C_N + K * C_K\right)}{\left(N + K\right)}$$

Note that removing images from the database will not affect the correlation matrix or the transform matrix derived from it. This inability to forget is common to many learning systems and in this particular application may lead to a decrease in accuracy of the search, especially if the content of the images removed differ greatly from added images. The risk of this actually happening during the use of the application in a practical situation is rather low. Nevertheless, a method for estimating the accuracy of the transform matrices is implemented and run on each update. This method works by multiplying the original transform matrix with the transpose of a transform matrix calculated from the new correlation matrix. If these two matrixes are identical, their product will be the identity matrix. To estimate how different the two matrixes are the sum of the absolute values of the off-diagonal elements is normalized by dividing it with the number of rows in the matrix. If this number is higher than a predefined threshold unique to each feature extractor, the user is asked to rebuild the database for the current feature extractor.

$$T_{org} * T_{new}^{T} = \begin{bmatrix} \tilde{I}_{1,1} & \tilde{I}_{1,2} & \cdots & \tilde{I}_{1,M} \\ \tilde{I}_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \tilde{I}_{M-1,M} \\ \tilde{I}_{M,1} & \cdots & \tilde{I}_{M,M-1} & \tilde{I}_{M,M} \end{bmatrix}$$

$$A = \sum_{i=1}^{M-1} \left| \tilde{I}_{i+1,i} \right| + \left| \tilde{I}_{i,i+1} \right| \Big/ M$$

8.6 Querying the database

The main feature of the application is of course searching for images. The querying method takes the path to the image, the feature extractor to be used and the number of results wanted. The query image's feature vector is calculated for the chosen feature extractor and compressed using the transform matrix. The search for the best mach then consists of a simple matrix multiplication of the query vector with the compressed matrix of all the images in the database. The resulting vector is then sorted and the top results returned.

Chapter 9
Implementing a new feature extractor

One of the main uses of the ImBrowse.NET platform is developing and implementing new feature extractors. In order to do so, the developer needs to alter the source code of the application as well as adding information about the new feature extractor in the applications configuration file. It is assumed that the developer has at least a basic understanding of the C# programming language, as well as the .NET 3.0 Framework.

The first file that the developer needs to edit is `FeatureExtractor.cs`. This file contains the definitions of all the feature extractor methods in the application. All feature extractors are implemented as methods that take a `string` containing the full path to an image and return an `array` of `doubles`. This array contains the uncompressed features of the input image and its length is defined by the developer. This is the programmatic equivalent of a feature vector. The following code shows a template for a feature extractor method:

```
public static double[] MyFeatureExtractor(string path) {
    double[] featureVector = new double[NUMBER_OF_FEATURES];
    byte[] pixelData = GetPixelData(path);

    return featureVector;
}
```

The template includes a call to the `GetPixelData()` method. This method returns an `array` of `bytes` containing the pixel data of the input image in BGR32 format. The array is one-dimensional, with the first four elements representing the *blue*, *green*, *red* and *alpha* value of the top left pixel, in that order. The following elements contain the pixel data on the top row and then continuing row by row from the left (see fig. 9.1). This method also has an overload that returns the height and width in pixels as `out` parameters if they are needed.

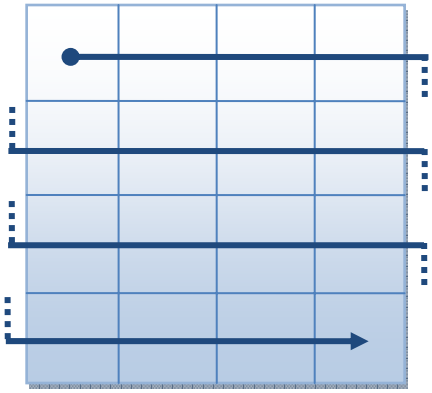

Figure 9.2: The pxel are mapped to the array starting from the top left

Other than the restrictions on the parameter and return value, the implementation of the feature extractor is left completely up to the developer. However, some pointers on how to access the pixel data might be helpful. To access the blue, green, red or alpha value of any given pixel, use the following code snippet:

```
byte blue = pixelData[(x + width * y) * 4];
byte green = pixelData[(x + width * y) * 4 + 1];
byte red = pixelData[(x + width * y) * 4 + 2];
byte alpha = pixelData[(x + width * y) * 4 + 3];
```

The variables `x` and `y` are the zero-based coordinates of the pixel in the original image and `width` is its width in pixels. In order to get a hold of the pixel width of the image, use the `GetPixelData(string path, out int pixelWidth, out int pixelHeight)` overload.

When adding a new feature extractor, a call to it must be added to the `ExtractFeatures` method. This method acts as a switch, controlling which feature extractor is called. The following is a template for the code the developer must add:

```
case "MyFeatureExtractor":
    featureVector = MyFeatureExtractor(path);
    break;
```

The `string "MyFeatureExtractor"` must match an entry in the applications configuration file (see below). The method that is called, here `MyFeatureExtractor`, should be the new feature extractor. It is recommended, but not technically necessary, that the string matches the name of the method.

After the feature vectors for all the images in the database have been calculated using the new feature extractor, the resulting matrix is compressed using *Principal Components Analysis*. The developer must therefore define how many eigenvectors to use. The optimal number differs depending on how the feature extractor operates, but a good rule of thumb is to use between 10-20 eigenvectors.

Before the feature extractor can be used, it needs to be added to the application's configuration file. The developer must add the name of his feature extractor method to the comma separated list of values for the `FeatureExtractor` key. The number of features the method extracts (i.e. the length of the vector), the number of eigenvectors used to store the PCA values and the accuracy tolerance for checking its transform matrix must also be defined.

Adding a method called `MyFeatureExtractor()` that has a feature vector length of 100, uses ten eigenvectors and has an accuracy tolerance value of 0.8 would look like this:

```
<add                                                    key="FeatureExtractors"
value="CalculateRGBHistogram,MyFeatureExtractor" />
<add key="NumbersOfFeatures" value="512,100" />
<add key="NumbersOfEigenvectors" value="20,10" />
<add key="TransformMatrixAccuracyTolerances" value="0.5,0.8"/>
```

9.1 Step-by-step Guide

The following is a compact step-by-step guide to adding a new feature extractor to the application.

1. Open the file called `FeatureExtractor.cs`.
2. Locate the commented code segment containging the template class `MyFeatureExtractor`. Copy the code in the template class and give it a unique name.
3. If your feature extractor doesn't need to know the dimensions of the image, uncomment the first call to the `GetPixelData()` method. If it needs to know the dimensions, uncomment the second call to `GetPixelData()` instead, as well as the declaration of the two integer variables `pixelWidth` and `pixelHeight`.
4. Decide the length of the feature vector, i.e. how many dimensions are needed to store the extracted data. Change the text value in the initiation of the `featureVector` array from `NUMBER_OF_FEATURES` to the number of features needed.
5. Perform the feature extraction using the pixel information in the `pixelData` array. Keep in mind that every four bytes represent one pixel in the BGR32 format, starting with the top left pixel and going row-wise down to the bottom right pixel. Also remember that in the BGR32 format, the first byte represents the blue value, the second the green value, the third the red value and the forth the alpha value.
6. Store the extracted feature values as double precision floats in the `featureVector` array. This is the last step in creating the new method.
7. Locate the definition for the class called `ExtractFeatures`. Copy the commented code segment containing the case template. Change the string in the case to the name of your new feature extractor. Also change the call to the method `MyFeatureExtractor` to a call to your new feature extractor method. Remember that C# is case sensitive.
8. Save and close `FeatureExtractor.cs` and rebuild the solution.
9. Open the file called `ImBrowse.exe.config`.
10. Find the key called `FeatureExtractors`. Add the name of your new feature extractor method to the comma separated value list of this key. Remember that this is case sensitive.
11. Find the key called `NumbersOfFeatures`. Add the length of the uncompressed feature vector of your new feature extractor method to the comma separated value list of this key.
12. Find the key called `NumbersOfEigenvectors`. Decide the number of eigenvectors needed for the compressed feature vector of your new feature extractor method. Add the number to the comma separated value list of this key. For more information on how to decide the number of eigenvectors, see the previous section.
13. Find the key called `TransformMatrixAccuracyTolerances`. Decide the tolerance threshhold for the transform matrix accuracy of your new feature extractor

method. Add the number to the comma separated value list of this key. For more information on how to decide this threshhold, see the previous section.

14. Save and close `ImBrowse.exe.config`.

Chapter 10
Interface

ImBrowse.NET is, as the name implies, written in .NET Framework 3.0. The interface is written using *Windows Presentation Foundation* (WPF). One of the main advantages of WPF is that it attempts to separate the UI from the business logic of an application. For a more detailed description of WPF, see section 3.1.

Due to time constraints, the original concept for the UI was never fully realized. However, the placeholder UI that is implemented still fulfils all the basic needs.

10.1 The Main Window

Once the application has started, the user is presented with the main window. From here he can choose a query image and see the matching results (see fig. 10.1). In the upper part of the left frame the controls for choosing query image, the maximum number of results and which feature extractor to use are found. Below the header information of the selected image is displayed (note that not all images contain such header data). In the right the best matches of the query are shown, with the best match in the upper left corner. Matching accuracy decreases row-wise, with the worst match found at the right end of the lowest row. At the bottom is a slider that lets the user zoom in and out of the result set.

10.2 Making a query

In order to begin a query the user must select a query image. This may be an image found in the database, but any image in the supported image formats will do. The supported formats are JPEG, Windows Bitmap, PNG, GIF and TIFF. Clicking the *Open* button will open a file dialog that lets the user choose an image (see fig. 10.2). The selected image is then displayed in the upper left corner of the main window. Next, the user needs to choose the desired number of results by typing it into the text box next to the *Open* button. The default value is 10, but any integer value will do. Note that displaying a very large number of images will require a lot of memory. Lastly, the user needs to choose which type of feature to use in the query by selecting the appropriate feature extractor from the adjacent drop-down menu. Changing feature extractor might take a few seconds as the appropriate data is read into memory. Note that in the applications default state only one feature extractor, the RGB-histogram is available. Pushing the *Query* button will execute the query. Depending on the complexity of the feature extractor, the size of the query image and the number of images in the database this might take a few seconds.

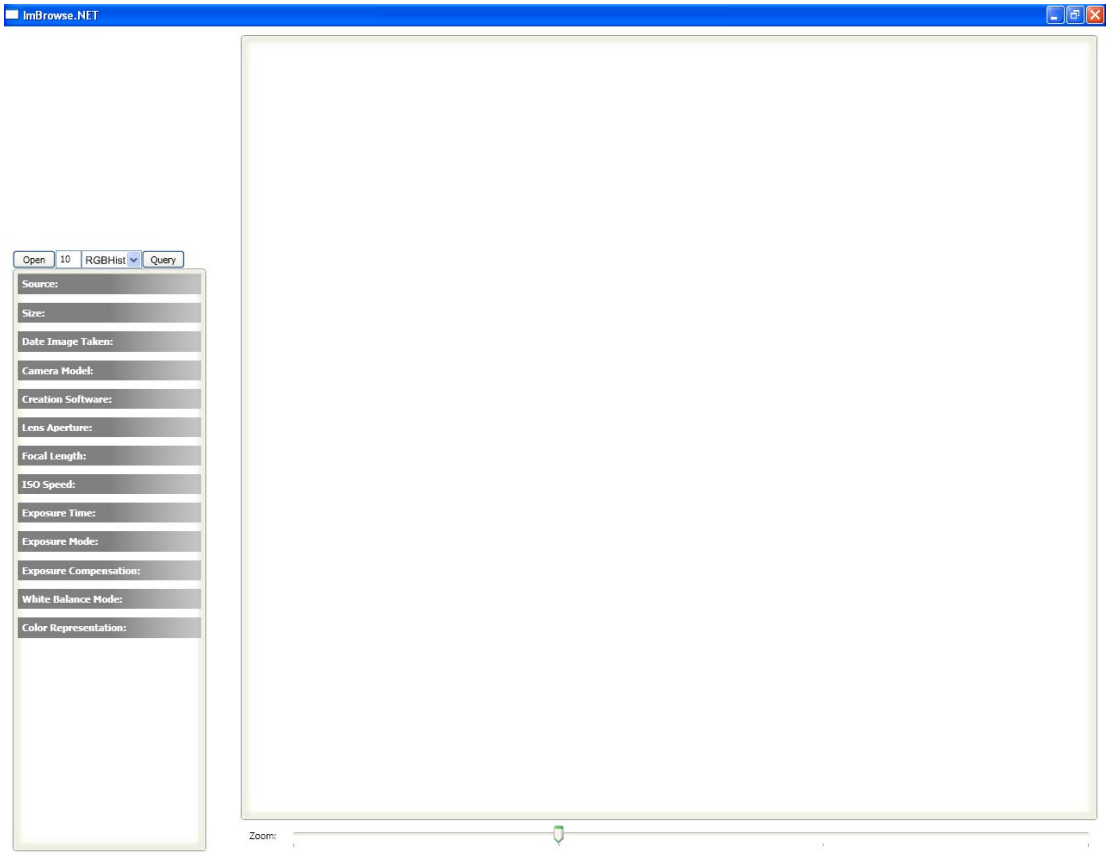

Figure 10.1: The Main Window

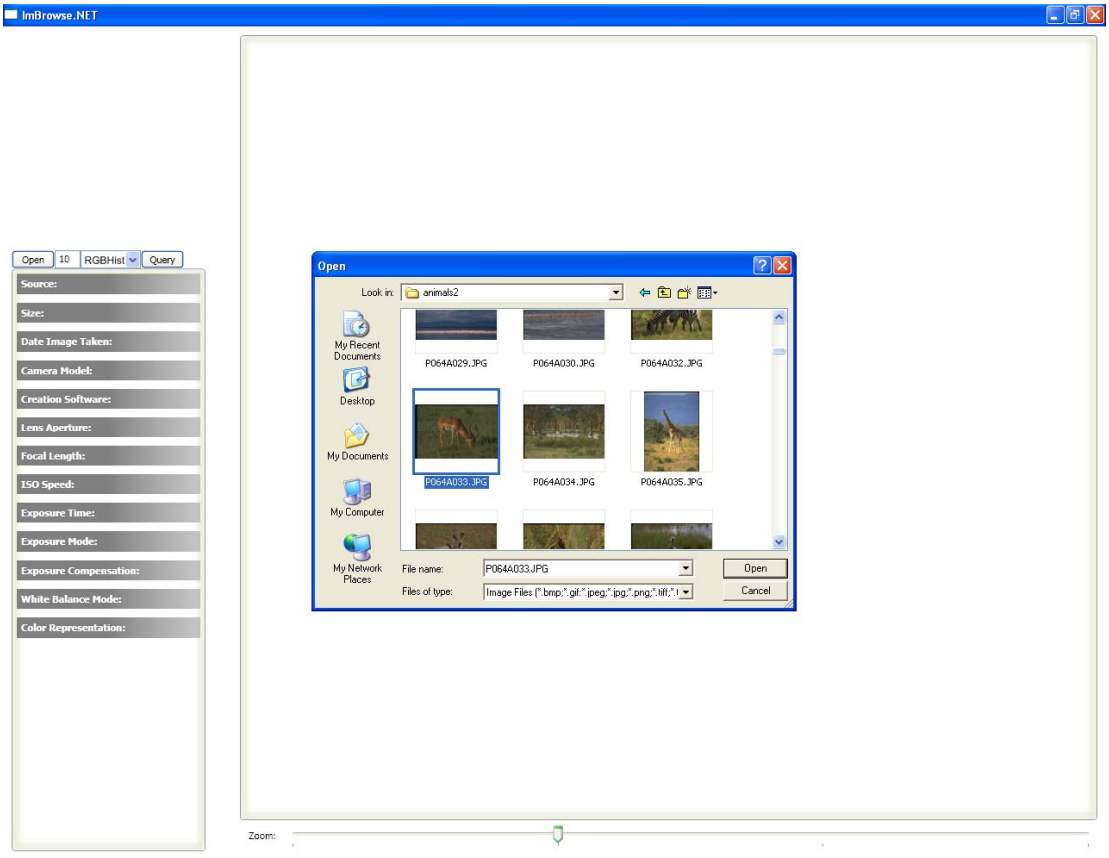

Figure 10.2: The open file dialog

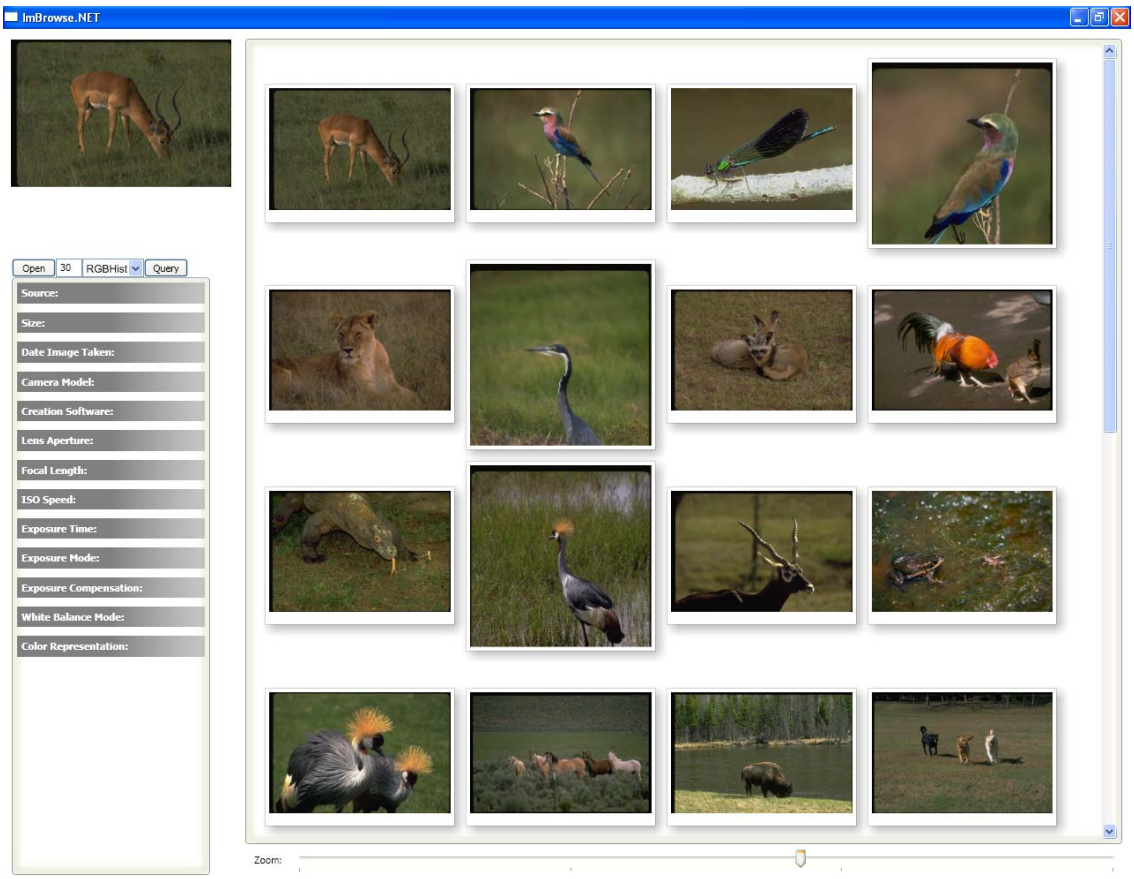

Figure 10.3: The result set zoomed in

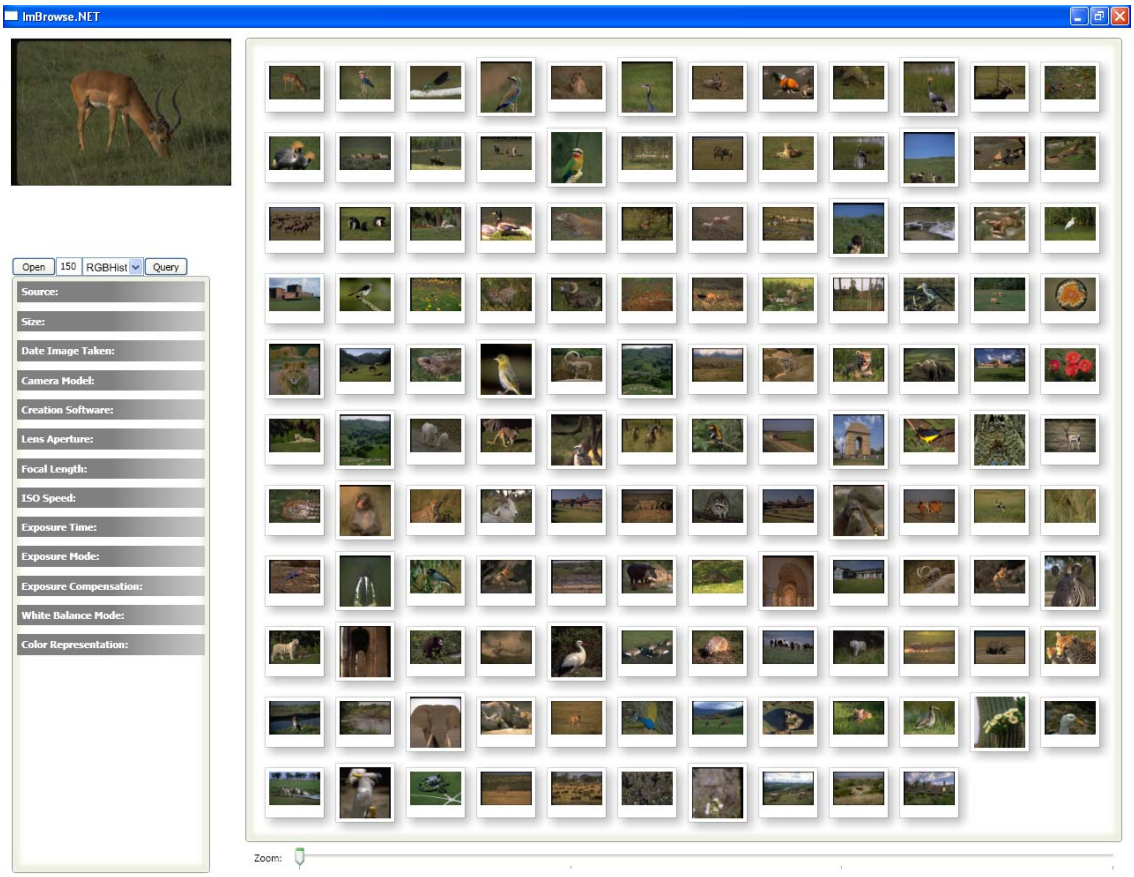

Figure 10.4: The result set zoomed out

10.3 Viewing the result set

Once the query is made, the best matches are shown in the right frame. By adjusting the slider below the result set, the user can seamlessly zoom in and out of the result set, displaying either a large number of images or a smaller number in greater detail (see fig. 10.3 and 10.4). In order to save memory, the images displayed in the result set are actually the thumbnail versions stored in the headers of JPEG images. This means that other image types as well as JPEG images that lack a thumbnail will only be displayed as text. A discussion on how to improve this so that previews can be shown for all supported image types can be found in section 11.1.

Right-clicking an image in the result set will bring up a context menu with two options: *Query* and *Edit*. Choosing *Query* will perform a new query with that image as input, using the selected feature extractor. *Edit* will open the *Photo Viewer* window.

Figure 10.5: Right-clicking opens the context menu

10.4 The Photo Viewer

The *Photo Viewer* is a window that shows an image in full resolution and provides some basic image manipulation tools (see fig. 10.6). The Photo Viewer is accessed either by double-clicking on an image in the result set or by right-clicking on it and choosing *Edit* from the context menu. The selected image will fill the width of the Photo Viewer window. Changing the size of the window will scale the image accordingly. There are three buttons at the bottom of the window: *Crop*, *Rotate* and *BlackAndWhite*. Clicking and dragging on the image will let the user select a section and *Crop* will crop the image to this section. *Rotate* will rotate the image 90° clockwise. *BlackAndWhite* removes all color information from the image, rendering it in grayscale. The user should note that these changes are permanent. These basic tools are legacy functions from an early rendition of the UI and should be removed or improved in future versions.

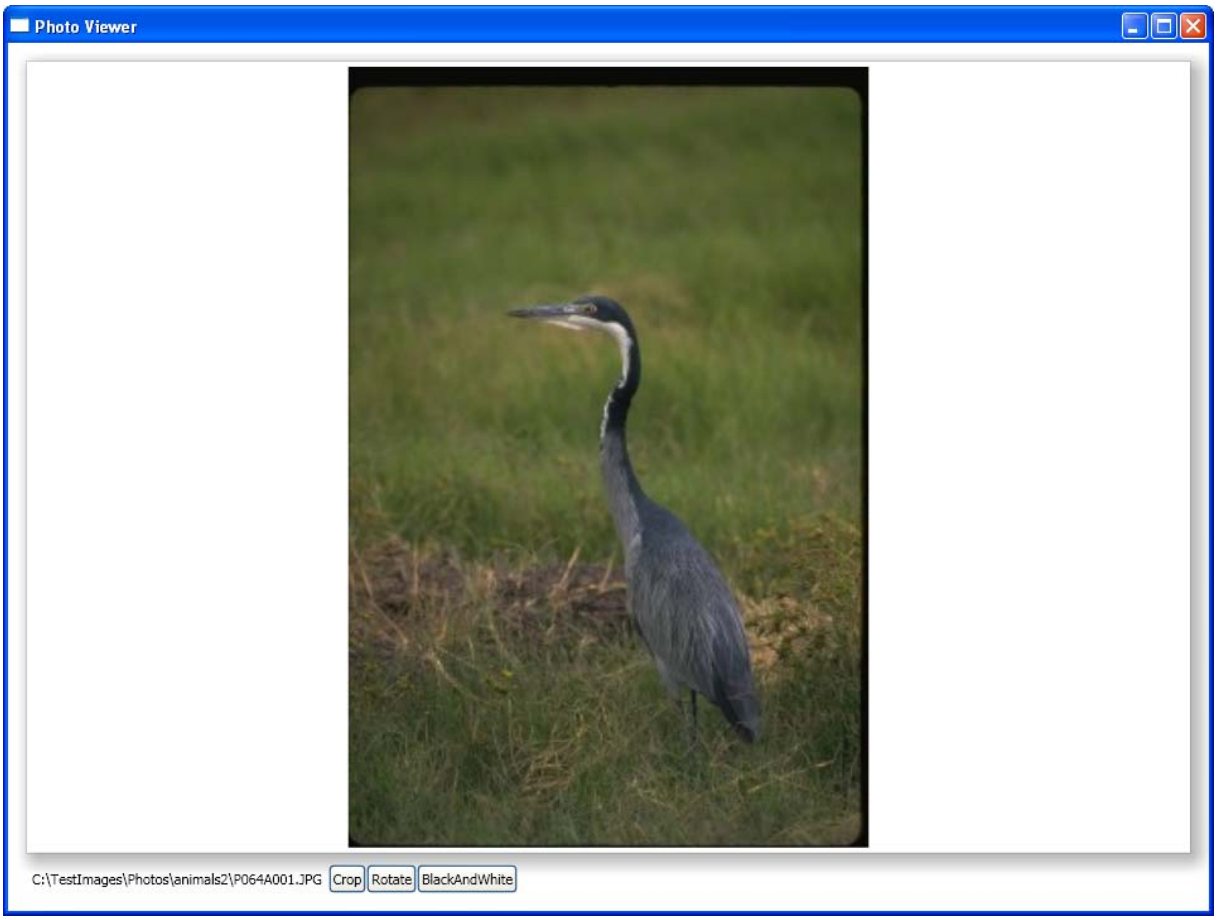

Figure 10.6: The Photo Viewer

Chapter 11
Conclusions

The goal was to construct a platform for the development and testing of feature extractors, to be used both in teaching and in more professional environments. The application works well for teaching purposes but needs more work, especially with the GUI. The modular architecture of the application makes it relatively easy to extend it with new query methods. Features such as an automatically updated database and PCA compression let's user focus mainly on the actual image processing code. The application is fairly fast and memory efficient compared to non-compiled solutions such as The MathWorks MATLAB.

11.1 Future work

The main purpose of ImBrowse.NET is to provide a platform for the development of feature extractors. As such, it is in its very nature to be extended and further developed. However, in addition to the implementation of new feature extractors there are other parts of the application that would benefit from additional work.

- The user interface was never fully completed. In particular, a way to set user preferences from within the GUI is lacking. All settings do now have to be set by manually editing the applications configuration file.
- Thumbnails are only shown for JPEG images. This is because the GUI uses the thumbnails contained in the headers of JPEG files not found in other formats. Using thumbnails generated by Windows Explorer instead could solve this problem.
- The distance measure is naïvely implemented as an *O(N)* solution that loops through all images in the database. Due to the PCA compression of the feature vectors, multi-dimensional indexing techniques should be able to reduce the time complexity to *O(log N)*.
- The exactness of the distance measure could be improved. Two solutions are implemented: a simple matrix multiplication and a squared distance measure. Neither of these has provided completely satisfactory results.
- The effectiveness of the function that measures the transform matrix accuracy has not been empirically proven. It also requires further testing in order to provide better guidelines for developers on how to choose the tolerance threshold for new feature extractors.

References

[1] http://desktop.google.com/dev/index.html

[2] Tran, L.V. (2003) *Efficient Image Retrieval with Statistical Color Descriptors*. Department of Science and Technology, Linköping University

[3] Fukunaga, K. (1990) *Introduction to Statistical Pattern Recognition*. Academic Press.

Appendix A
Creating your first feature extractor

In the following exercise you will learn how to create your very own feature extractor in ImBrowse.NET. You will write your own extraction method and integrate it with the application by adding it to the configuration file.

Step 1: Concept

The first thing we need to decide is what type of feature we are interested in. For this exercise we will choose the simple histogram. We will use a bin size of 1, giving each gray value its own bin. With an 8-bit image that will give us 256 bins, which means the feature vector will have a length of 256. We also need to decide how many eigenvectors we want to use to represent our feature vector after compressing it with PCA (Principal Components Analysis). This number is best achieved by testing, but using 20 eigenvectors should give us an adequately good representation. Using fewer eigenvectors will reduce the size on disk of the stored features and increase query speed at the cost of accuracy. Finding a balance between these factors is an important part in creating a new feature extractor.

Step 2: Creating our method

Now that we know what we want to do we are ready to start writing some code. The first thing we need to do is download the source code for the ImBrowse.NET application. A RAR-archive containing all the code can be found here:

http://www.itn.liu.se/~reile/ibn

Once the archive has been unpacked, open the Visual Studio solution file `ImBrowse.sln` by double-clicking it. When Visual Studio has started and the solution has loaded, locate the *Solution Explorer* on the right hand side of the screen. If the Solution Explorer is not open, go to *View/Solution Explorer* or press Ctrl+Alt+L. In the Solution Explorer window, double-click on the file called *FeatureExtractor.cs*. This is the only file we need to edit. Now we're ready to start writing our code.

Locate the commented code segment marked `Feature Extractor template`. Copy and paste all the code in this method. First, let's give our method a little more descriptive name. Change the name of the method from `MyFeatureExtractor` to `CalculateHistogram`. Next we need to decide which of the two calls to the

`GetPixelData` method we want to use. Since we don't need to know the dimensions of an image to calculate its histogram, we'll use the first call. Note that while we do need to know the number of pixels in the image in order to normalize the histogram, the relation between the height and width are not needed. Go ahead and uncomment the first method call. It should look like this:

```
byte[] pixelData = GetPixelData(path);
```

This will give us an array containing all the pixel data of the image in BGR32 format. Now we need an array to store our feature vector in. You can find this declaration at the top of the method. All we need to do is change the text `NUMBER_OF_FEATURES` in the initiation to the length of our feature vector, which we decided to be 256. The declaration should now look like this:

```
double[] featureVector = new double[256];
```

Now we're ready to start calculating the histogram. In order to do so we need to loop through all the values of the pixel data array. The BGR32 format that this array uses represents each pixel as four 8-bit integers, one for each color channel and one for the alpha channel. The order is blue, red, green, alpha. Let's start by making a loop that takes steps of four through the entire array:

```
for (int i = 0; i < pixelData.Length; i += 4) {
}
```

For every pixel we need to calculate its intensity and place it in the corresponding bin in our feature vector. We do this by adding the weighted values of the color channels (ignoring the alpha channel), using the standard conversion of RGB to grayscale (i.e. 30% red, 59% green, 11% blue). We'll store this in a variable. Add the following line below the declaration of the other method variables:

```
byte bin;
```

Add this line to the for-loop:

```
bin = (byte) (pixelData[i] * 0.11 +
              pixelData[i + 1] * 0.59 +
              pixelData[i + 2] * 0.30);
```

Remember that the order of the color channels in the BGR32 format is backwards. Now that we know which bin the pixel belongs in, we need to increase it by 1. Add this line to the for-loop:

```
featureVector[bin]++;
```

All we need to do now is normalize our histogram. To do that we need to divide the feature vector by the number of pixels in the image. We can find the number of pixels by dividing the length of the pixel data vector by 4. Add the following loop after the first one:

```
for (int i = 0; i < featureVector.Length; i++)
    featureVector[i] /= (pixelData.Length / 4);
```

Your method should now look like this:

```
public static double[] CalculateHistogram(string path) {
    double[] featureVector = new double[256];
    byte[] pixelData = GetPixelData(path);
    byte bin;

    for (int i = 0; i < pixelData.Length; i += 4) {
        bin = (byte) (pixelData[i] * 0.11 +
                      pixelData[i + 1] * 0.59 +
                      pixelData[i + 2] * 0.30);
        featureVector[bin]++;
    }

    for (int i = 0; i < featureVector.Length; i++)
        featureVector[i] /= (pixelData.Length / 4);

    return featureVector;
}
```

Now we need to add our new method to the list of methods found in the `ExtractFeatures` method. Locate this method, and copy and paste the commented code segment in the switch block. All we need to do is to change the string in the case argument and the method call to the name of our method.

```
case "CalculateHistogram":
    featureVector = CalculateHistogram(path);
    break;
```

Save your changes and build the solution by selecting *Build/Build Solution* or by pressing Ctrl+Shift+B.

Step 3: Editing the configuration file

Our new feature extractor is done! We now need to let the application know it has a new query method by editing its configuration file. Open the file called *ImBrowse.exe.config* found in the *bin/Debug* folder. You can open it in Visual Studio, but any text editor will do. This is an XML-file specifying the applications user settings. We need to edit four keys: `FeatureExtractors` contains a list of all implemented feature extractors; `NumbersOfFeatures` lists the length of the corresponding feature vectors; `NumbersOfEigenvectors` specifies the number of eigenvectors used to represent the feature vectors after PCA compression; `TransformMatrixAccuracyTolerances` contains threshold values for a method that is used on database updates. To the first three keys we add the name of our new method, the length of its feature vector and the number of eigenvectors we want to use, which we chose to be 20. To the last key we'll add the number 0.5. This number is only used when the database is updated without completely rebuilding it and we don't have to worry about that now. The keys should now look like this:

```
<add          key="FeatureExtractors"          value="CalculateRGBHistogram,
CalculateHistogram" />
<add key="NumbersOfFeatures" value="512,256" />
<add key="NumbersOfEigenvectors" value="20,20" />
<add key="TransformMatrixAccuracyTolerances" value="0.5,0.5" />
```

The first values for the keys are for the RGB-histogram method already implemented.

Step 4: Build the database

The only thing left to do is to rebuild the database. To do this just run the application by double-clicking on *ImBrowse.exe* found in the *bin/Debug* directory. Note that this might take a few minutes. Once the application starts you are ready to start searching for images using your new histogram feature!