Web-based Image and Video Navigation

Final Report

Alex May
adm00@doc.ic.ac.uk

June 16, 2004

Supervisor: Stefan Rüger
srueger@doc.ic.ac.uk

Second Marker: Ian Harries
ih@doc.ic.ac.uk

Abstract

Content-based image and video retrieval has always been a challenging problem in computer science. In response, many techniques have been developed to exploit both the visual and semantic similarity between images. These include feature-based searching mechanisms and the creation of novel browsing structures.

In this project, we present a web-based image browser and search engine application. We consolidate paradigms of text-based search, content-based search with relevance feedback, hierarchical browsing, lateral browsing, temporal browsing and historical browsing into a unified interface. By providing tight integration of techniques and a rich set of user interactions we are able to equip the user with substantial navigational power.

Project homepage and demonstration:
http://www.doc.ic.ac.uk/~adm00

Code archive:
/homes/adm00/public_html/ibase_code

Acknowledgements

I would like to thank Dr. Stefan Rüger for his supervision, feedback and encouragement throughout the project. It has been a great pleasure to benefit from his experience and insight in the field and he has been a great source of inspiration. I would like to thank Ian Harries for his invaluable feedback and suggestions during the early stages of the project.

My utmost thanks also go to Daniel Heesch. This project builds upon much of his research, and I have learned a great deal from his work. Our discussions were sometimes lengthy and I express gratitude for the time and energy he has invested.

Finally, I would like to thank all those who helped with evaluation: Xia Wu, Tristan Carmichael, Eleanor Davies, Phil Carmalt and Jing Zhang. Their participation was crucial to the insightful evaluation of system performance and usability.

Contents

Figures

1 Introduction

1.1 Content-based image retrieval

Images and video have become commonplace on the WWW and in many multimedia applications. This has given rise to large digital image and video databases that need to be searched effectively and on demand. Manually classifying or annotating images is laborious and relies on the judgement of experts. Even if a database is manually annotated with keywords, it can be difficult to formulate a text query which describes exactly what is sought. Clearly there are many benefits of a content-based image retrieval (CBIR) system. That is, a retrieval method based upon features that are automatically extracted from the images themselves. There are many challenges involved in developing CBIR techniques and the area has received much attention from the research community in recent years.

By dissecting video into keyframe *shots* we can use the same strategies in approaching content-based video retrieval as we can in content-based image retrieval [1]. We shall assume the equivalence of the two problems based upon this principle. Video has the advantage that textual data can be automatically obtained from subtitles and speech recognition transcripts, rather than manual annotation.

There are many commercial and research applications of CBIR [2], including those in the areas of:

- Security and defence, e.g. fingerprint matching
- Intellectual property, e.g. trademark databases
- Media archives, e.g. television footage, art galleries
- Scientific imaging, e.g. medical imaging, weather forecasting

Several previous studies of CBIR such as [3] have found that user's search strategies fall into several main types:

- Search by association – when there is no specific aim other than to find items of general interest, analogous to 'surfing' the web
- Search for a class of image – when the aim is to find images that fulfil certain criteria, e.g. all pictures of animals
- Search for a specific image – when the user has a particular image in mind, e.g. a picture of Tony Blair

Often, the CBIR techniques applied in a system are predominantly suited to fulfilling the requirements of only one or two of these types of user.

1.2 The challenges

The challenges of CBIR lie in two main areas [4].

Firstly, there is a problem of deriving high-level image meaning from primitive image features such as texture, colour and shape. It is apparent that the raw image data alone is not sufficient, but that a large amount of prior world knowledge is required. This is commonly referred to as the 'semantic gap'.

Secondly, there is a problem resulting from the semantic ambiguity of images. Each image might have a number of possible high-level representations, and there is no way of knowing which the user has in mind. This problem is commonly referred to as 'polysemy'.

In tackling these challenges, most of the main CBIR systems have focused research on the set of features extracted from the images, and there has been little emphasis put on user interaction, except allowing some system configuration such as defining the weights of features used for retrieval.

For a user to determine the relevance of an image requires a relatively small mental load compared to, for example, text documents. This would suggest that user-interaction (compared with automated retrieval) has a more important role to play in CBIR than in text document retrieval, and deserves greater attention from the research community.

1.3 Project aim

This project proposes that there are advantages in the seamless integration of a number of searching and browsing techniques that have independently proved successful in the past. By increasing the level of user interaction and feedback into the searching process, we believe that relevant images can be located in large collections quicker and easier than ever before.

Section 2 presents background information to CBIR searching and browsing techniques and Section 3 outlines a specification of what is to be achieved. Section 4 discusses the interface design decisions made and Section 5 describes implementation methods used. Section 6 provides some initial quantitative and qualitative evaluation and in section 7 we reflect on the achievements made and the contributions of this project to the image retrieval research community.

2 Background

2.1 Searching

A traditional CBIR search engine involves the following steps:

- Feature extraction for each image in collection (pre-computed)
- Feature extraction for query image(s) (real-time if not in collection already)
- Calculation of the distance of each image in the collection from the query image, based on a combination of the distance values computed for each feature descriptor (real-time)
- Results returned ordered by lowest distance (real-time)

Figure 1 shows two common processes used for searching in traditional CBIR systems. Results from the manual process are wholly reliant on the performance of the system. Results from the interactive process are a consequence of an iterative procedure involving the user modifying the query based upon the system results. This allows the system to draw upon the user's prior knowledge of the world and hence understanding of the high-level meaning of an image, and their knowledge of the topic from which the query was derived. There has been much research into the best way of effectively integrating this feedback from the user into the search process. Examples of techniques include *interalia* supervised learning prior to retrieval [5], interactive region segmentation [6], and interactive image database annotation [7]. We shall focus on one of the more effective methods called *relevance feedback*.

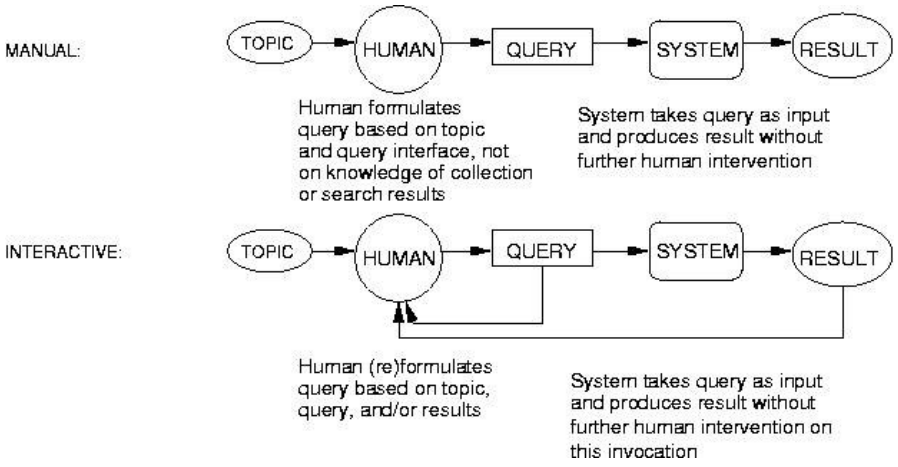


Figure 1. The searching process [8]

2.1.1 Query generation

The problem of automated topic analysis and query generation is beyond the scope of this project. We shall assume that the user has enough understanding of the topic to generate a query using one or more example images along with textural input. Another possibility for providing the query includes giving a rough sketch of the image sought [9].

2.1.2 Feature extraction

Much scrutiny has been placed on the feature set used as image descriptors in the search engine. The optimal combination of features differs depending on the type of image collection. Certain features are better at describing the semantics of certain types of image. For example, a Colour Histogram descriptor would be useless when describing similarity between images in a collection of black and white sketches. Because this project is largely concerned with the user-interaction side of CBIR, we shall only provide an overview of some of the commonly used features in current.

Most features are based upon extracting colour, shape and texture information. The following are used for video and photo retrieval. See [10] for more information.

HSV Global Colour Histograms – This histogram descriptor involves quantising the distribution of pixels in an image in the HSV (hue, saturation and brightness) colour space. The histogram is then normalised so that all bins add up to 1.
HSV Focus Colour Histograms – This method applies the HSV colour histogram technique to the central 25% of pixels. Hence this can be used to disregard the background, and find similarities between the objects of focus.
Colour Structure Descriptor – This method uses the HMMD (Hue, Min, Max, Diff) used in the MPEG-7 standard. To encapsulate local image structure, an 8x8 window is slid over the image. This allows the descriptor to distinguish images with the same global colour distribution but a different local colour distribution.
Marginal RGB Colour Moments – This method involves taking histograms for each of the RGB colour channels, and computing the mean, second, third and fourth central moments.
Convolution filters – Here, Tieu and Viola's [11] method is used to generate 25 different feature maps based upon applying a set of 25 primitive filters to the grey level image.
Variance – The variance feature is based upon taking the standard deviation of grey values within a sliding window of 5 x 5.
Uniformity – This is a statistical measure based upon the frequency of pixels with a particular grey level in 8 x 8 tiles.

Smoothness – This is another statistical measure based upon the variance of particular grey values, again in 8 x 8 tiles.
Thumbnail – The thumbnail descriptor scales down the image to 44 x 27 pixels and records the grey values of each pixel. This helps identify near-identical copies of images.
Text – By using speech recognition transcripts, the Managing Gigabytes [12] search engine is used to annotate each image. This allows the images to be searched effectively by text as well as by example.

The following are used for sketch retrieval. See [13] for more information.

Chain Code Histograms – This is a method of deriving a description of the sketch contour by reading in the contour in a clock-wise or counter clock-wise direction and recording the relative position of each pixel to its neighbour. This can then be encoded to a representation which is invariant of translation, rotation and scaling.
Moments – The moment of an image can be used to capture distributional properties. It is calculated in a similar way to mechanical moments. The first order moment (p=1, q=1) are for an X by Y size image can be calculated as follows:

$$m_{pq} = \sum_{i=1}^{x_{\max}} \sum_{j=1}^{y_{\max}} i^p j^q f(i, j)$$

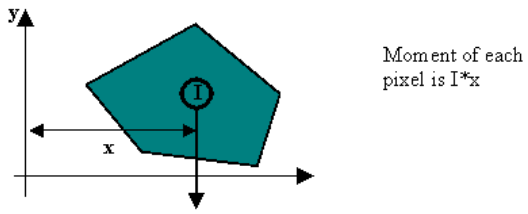


Figure 2. Taking first moments about y [14]

Central moments can be used to provide translation invariance.
Fourier Descriptors – A Fourier descriptor can be generated by obtaining a periodic function from the contour pixels. This periodic function can be expanded into its sinusoidal components. The low frequency components are of most interest, since these describe the general shape of the object.

2.1.3 Retrieval

Retrieval can be carried out using the k-nearest neighbour approach, by providing positively and negatively classified examples, and classifying test images according to their proximity to these. The following is an extract from [10].

We use a variant of the distance-weighted k-nearest neighbour approach. Positive examples are supplied by the user, and a number of negative examples are randomly selected from the database. The distances, for descriptor d, from the test image T_i to each of the k nearest positive or negative examples (where 'nearest' is defined by the Euclidean distance in feature space) are determined, and a distance measure d calculated as follows:

$$d_f(Q,T_i) = \frac{\sum_{n\in N} (dist_f(T_i,n)+\varepsilon)^{-1}}{\sum_{q\in Q} (dist_f(T_i,q)+\varepsilon)^{-1} + \varepsilon},$$

where Q and N are the sets of positive and negative examples respectively amongst the k nearest neighbours, such that $|Q| + |N| = k$. ε is a small positive number to avoid division by zero. Images are ranked according to the convex combination

$$D_s(Q,T_i) = \sum_f w_f . d_f(Q,I_i),$$

where w_f is the weight of descriptor f subject to $\sum w_f = 1$ and $w_f \geq 0$.

2.1.4 Relevance feedback

Relevance feedback is a method of improving the performance of a search by increasing the level of user-interaction. After the user has formulated a query, and the CBIR system has retrieved a set of results, the user can provide information about which of the images in the results are relevant and which are irrelevant. This can be used to give the system a better indication of the weight vector to use when combining features.

One implementation of such a technique is given in [2]. This uses a user-interface in which the thumbnails that are returned from a search are displayed such that their respective distance from the centre of the screen is proportional to the dissimilarity $D_s(Q,T_i)$ of the thumbnail Ti to the query set Q. The user then provides relevance feedback by moving relevant items closer to the centre of the screen and irrelevant items to the outside of the screen.

Hence the user provides a vector of distances $D_u(Q,T_i)$ which differs from that computed by the system.

We then find a new weights vector w_f for which the sum of squared errors

$$SSE(w) = \sum_{i=1}^{N} [D_s(Q,T_i) - D_u(Q,T_i)]^2 = \sum_{i=1}^{N} \left[\sum_f w_f . d_f(Q,T_i) - D_u(Q,T_i) \right]^2$$

is minimised under the constraint of convexity. We can then repeat the search to retrieve a new set of results using the new weights vector. [2] found that the boost in performance provided by relevance feedback levelled off after about 4 iterations, leaving a 20% improvement in mean average precision.

2.2 Browsing

As well as searching for images by providing text and example images, we shall also provide several methods of browsing the image archive. Browsing is the traditional method of a finding an image in a collection when the images have been manually sorted or categorised. However, it has been shown that browsing is an effective method even when the browsing structure has been automatically generated [10].

The benefits of browsing arise from the human ability to quickly and accurately assess the relevance of a set of images – much more accurately than any automatic feature set comparison. The task lies in structuring the search space such that the set of images the user has to assess is sufficiently narrow. Five types of browsing have been outlined below.

2.3 Types of browsing

2.3.1 Hierarchical

Hierarchical browsing is used to navigate many traditional image collections, where the images have been manually sorted into categories. The browsing structure is often derived from the underlying file structure of the image collection. Indeed, an image browsing function is included within the file exploring capabilities of recent operating systems. Figure 3 shows two examples of traditional hierarchical image browsers. Many web-based collections employ a simple hierarchical structure such as BINS [15]. The browsing structure provides a fast and efficient method to locate an image in a collection.

Hierarchical browsing is suited to small collections which can be categorised easily. However, a large collection of, say, video keyframes requires much manual work to categorise the images before this method can be used.

Most hierarchical browsers tend to have a tree structure user interface, which clearly shows the hierarchical organisation. This allows the user to expand and collapse parts of the tree/hierarchy as they browse through the images. Tree interfaces have been used for years to browse hierarchical file structures in WIMP operating systems.

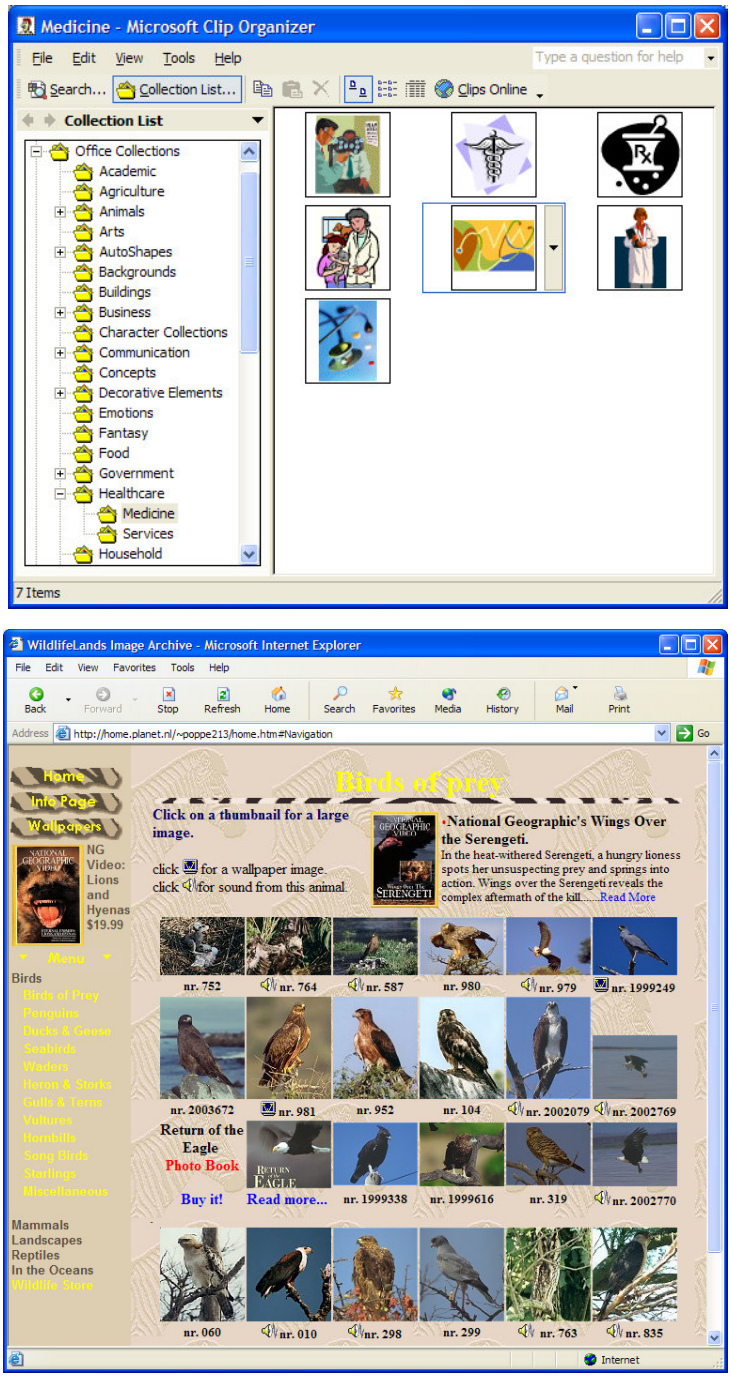


Figure 3. Hierarchical browsing examples: WildlifeLands Image Archive [16] and Microsoft Clip Organiser [17]

2.3.2 Temporal

Temporal browsing arises from the concept that each image in a collection has a predecessor and a successor image. This is suited to browsing a video archive, since a keyframe's neighbours can be determined from the video sequence. By storing this information offline in an index file, we enable the user to easily browse backwards and forwards in the video sequence at runtime.

If a particular keyframe is deemed by the user as relevant then it is likely that the neighbours of that image will also be relevant, since they are likely to be keyframes within the same video sequence. Hence this method is valuable when searching video archives.

Its applicability to a pure image collection is less clear, however. It is possible this technique is still of use when some temporal or, more generally, sequencing information is available. An example of such information might be timestamps applied to photographs by a camera. In this case, temporal browsing could be used to browse the photos in chronological order.

Figure 4 shows a content-based video browser which employs both hierarchical browsing and temporal browsing.

Figure 4. The IDIAP Video Browser [18]

2.3.3 Lateral (NN^k Networks)

Lateral browsing is a fresh approach to CBIR and has proved a useful additional browsing structure [19]. The idea of lateral browsing is borne out of several limitations with the traditional query by example approach to searching. These include the inability to navigate across the results provided by different weightings of features (the 'feature space') and the fact that time complexity increases linearly with collection size. Below I have summarised the mechanism, as described in [19].

Lateral browsing seeks to overcome these problems by determining the set of images that could be retrieved for a particular example image using any combination of features. We define the similarity between two images Q and I to be the weighted sum over the feature specific similarities:

$$S(Q,I) = \sum_f w_f F_f$$

where w_f is the weight of descriptor f subject to $\sum w_f = 1$ and $w_f \geq 0$. On this principle, we can build up a network whereby image Q is connected to an image I if there is at least one instantiation of the weight vector w such that it causes the image I to have the highest similarity *S(Q,I)* among all images of the collection.

This set of images can be pre-computed for each image in the collection. For each image, we store a set of images that were retrieved top under some feature regime and the number of times this occurred. In this way, we expose the polysemy involved in image retrieval. Because we can compute and store this network structure (dubbed an NN^k network), the user can navigate the collection through this network in real time, regardless of the size of the collection. The network produced was found to have properties that balance randomness (far-reaching arcs to other parts of the network) and high regularity (similar content is in close proximity to the current image). This makes it ideal for organizing and navigating through image information.

[19] suggests an implementation in which the user can be provided with access to the network without the need for a prior search through a clustering of the high-connectivity nodes. When the user clicks on a particular image, they are presented with that image in the centre of the screen and the network of neighbours drawn around it (Figure 5). By clicking on a neighbour, this node is transferred to the centre, and its neighbours are drawn accordingly. Thus the user can follow relevant links in the network to browse for an image.

2.3.4 Historical

Many recent graphical applications have included a function to maintain a history of events, and allowing the user to navigate backwards and forwards through their events. The classic example is a WWW browser (Figure 6). The browser maintains a list of pages visited. At any point, the user can browse back to the previous page viewed and forward again if necessary. This allows the user to retrace their steps if they make a mistake in their browsing path, instead of having to start over.

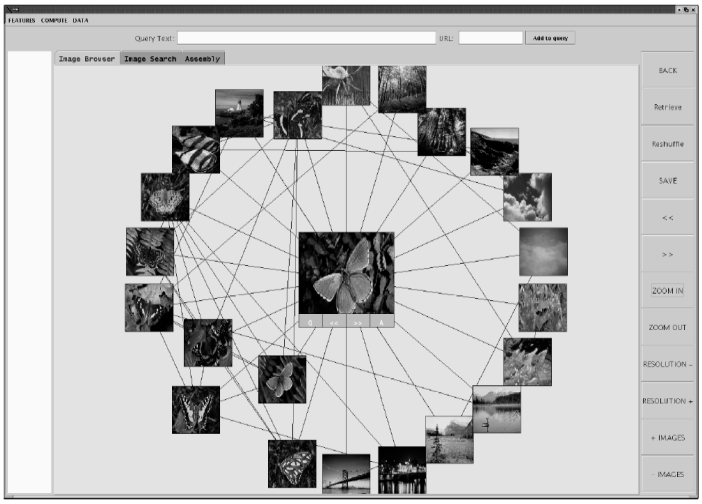


Figure 5. NN^k network shown around the chosen butterfly image

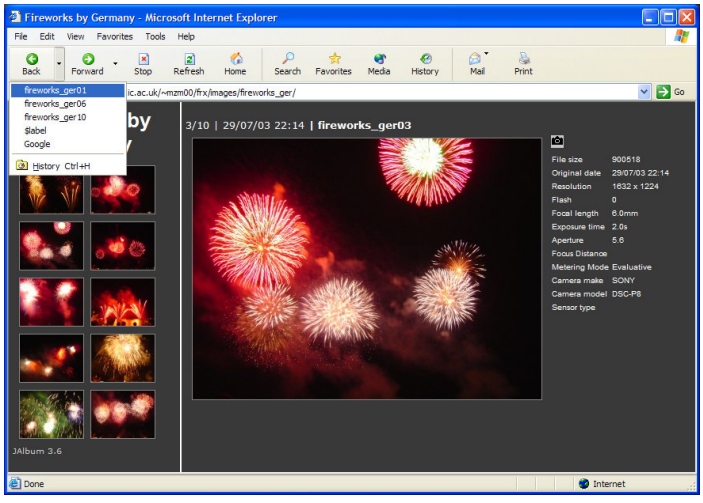


Figure 6. MS Internet Explorer displaying a hierarchical image collection

2.3.5 Search results

Another type of browsing that should be considered is browsing the results of a search. Many traditional search engines such as the Google Image Search [20] have used a simple line-by-line system, where the results are ranked from best in the top-left corner to worst in the bottom-right. Thus the user would scan the results in a similar way to reading a piece of English text.

A novel approach employed in the *Aetos* browser [10] is to return the results in a spiral, where the distance to each image from the centre is proportional to its dissimilarity to the query as computed by the system. This layout facilitates the relevance feedback implementation employed.

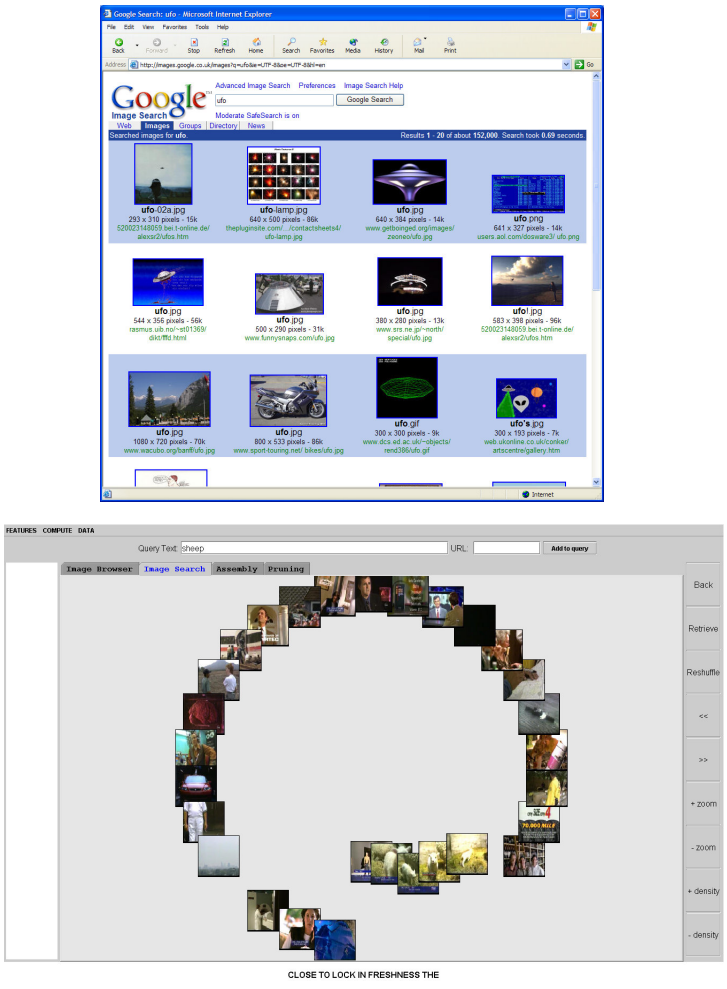


Figure 7. Google Image search [20] and Aetos browser [10]

2.4 The State-of-the-art

Image retrieval requires research in multiple different disciplines. This is especially true for the areas of computer vision responsible for describing images in terms of a number of features. Some examples of different techniques used in recent research include [4]:

- grouping
- edge extraction
- image segmentation
- texture features
- wavelets
- probabilistic matching
- histograms

As previously mentioned, there has been some effort in recent years in improving the level of user-interaction and improving the user-interface. Notable examples from TRECVID 2003 include:

- Video Browsing and Retrieval System (VIRE) [21] - The MediaTeam Oulu and VTT TRECVID experiments provided a novel approach using cluster-temporal browsing. This combines timeline presentation of video with videos with content-based retrieval. This reduced the effect caused by ambiguous results usually obtained from a traditional content-based example search.
- Interactive search using indexing, filtering, browsing and ranking [22] - This Mediamill and Univ. of Amsterdam team presented a four-step process for interactive search in which the user performs a traditional content-based search, and then browses the results to build up a set of ranked images.
- CDVP Dublin City University - Interactive Search Task Experiments [23] - This team employed a method in which results are presented as a group of five sequential shots, shown with their associated text. The user can then choose to save a particular image or add it to the query. The user must select the relative importance of the image in the query.
- ViewFinder [24] – The team from Indiana University demonstrated a user interface in which 8 results are returned in rank order. The user can then choose to view details about a particular image, or provide feedback by 'promoting' an image. This retrieves keywords associated with that shot includes them in the next search iteration. They also employ a video browsing mechanism using a combination of the date and video source.

2.5 TRECVID

The TRECVID [25] conference series is sponsored by the US National Institute of Standards and Technology. It aims to encourage research in automatic segmentation, indexing, and content-based retrieval of digital video. TRECVID provides a large test collection, uniform scoring procedures, and a forum for participants to compare their results. Imperial College has participated in several areas of the TRECVID evaluation in the past [1, 10]. The search task of TRECVID has provided a useful way to measure the effectiveness of new research into searching and browsing strategies and also provides an indication of the current state of research from the world's leading organisations in the field.

The search task for TRECVID 2003 was as follows: "Given the search test collection, a multimedia statement of information need (topic), and the common shot boundary reference for the search test collection, return a ranked list of at most 1000 common reference shots from the test collection, which best satisfy the need". More details of the restrictions for the task are included in the guidelines [8]. It is expected that the search task will not change significantly for TRECVID 2004.

3 System Specification and Prototype

The overall aim of the project is to create a web-based image and video browser for content-based retrieval that integrates the current research into browsing and searching into a unified interface. The resulting browser should provide the user with the power to browse a large image collection quickly and accurately. We shall build upon the work presented in [10] to enable us to concentrate on the important issues of integrating the searching and browsing mechanisms, and improving the user-interface.

3.1 System overview

The browser and search engine should be implemented under a client-server model. This enables us to centralise the search engine backend and the interface to the image collection. Figure 8 shows the suggested system architecture for a web-based model:

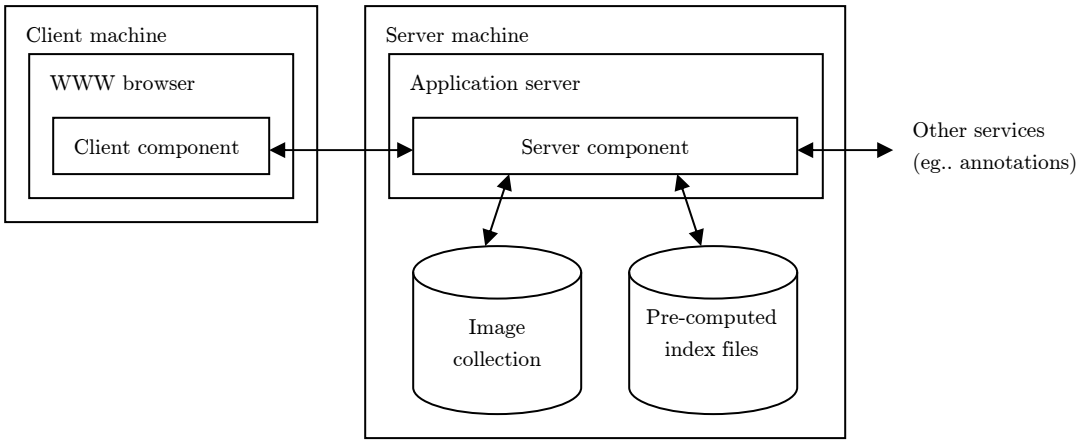


Figure 8. Suggested system architecture

3.2 Image and video collections

The browser will primarily be used with three image collections:

- Corel image collection (6192 images with category information)
- Sketch collection (238 black and white images)
- TRECVID test collection (32318 keyframes of video)

The differences between these collections should aid evaluation of the browser. If time permits, then the possibility of applying the searching and browsing mechanisms to a new collection could be considered.

3.3 Pre-computed data

3.3.1 Search

To reduce the calculation that has to be done in real-time, we can pre-compute much of the data involved in searching and browsing. We can pre-compute the feature descriptors for each image in the collection and save the results to disk in the same structure as the images themselves. Hence to compare the similarity of two images at runtime we simply need to compare the saved feature descriptors rather than compute them on the fly.

3.3.2 Hierarchical browsing

The hierarchical structure of the images is preserved in the file structure of the collection. Having been pre-categorised by an expert, the user can browse the hierarchy in the browser at runtime.

3.3.3 Lateral browsing

The NN^k network structure should be pre-computed and stored offline. This is represented by an index file for each image, containing that image's nearest neighbours. These files are used at run time to draw the lateral browsing structure.

3.3.4 Temporal browsing

The temporal sequence of images should be stored as an index file containing a list of the images along with their predecessor and successor as defined by some criteria, such as order in a video sequence. This file can be used at runtime to perform temporal browsing.

3.4 User interface (client-side)

The user interface should seamlessly integrate the following main features:

- Search with relevance feedback, browsing results by rank
- Hierarchical browsing
- Lateral browsing using NN^k network
- Temporal browsing

- Historical browsing
- Image/video viewer

Figure 9-Figure 12 demonstrate a prototype implementation that might satisfy these requirements. It is based upon the notion that at any time in the browser, we have an 'image of interest'. We could then provide a number of tabs for browsing in different dimensions with respect to this image. For example, if the user starts browsing using the hierarchy of categories, and then wishes to proceed browsing using the NN^k network, they can select the image in the hierarchical browsing tab, and this will become the centre of interest in the lateral browsing tab. In this way, we can maintain consistency across all the tabs, and thus provide different 'views' upon the same information. Below we outline what effect changing the image of interest would have on each of the views:

- Search with relevance feedback, browsing results by rank – if the image of interest is among the search results, then it would be selected, otherwise there is no change.
- Hierarchical browsing – the category in which the image of interest resides is shown. The image of interest is selected.
- Lateral browsing using NN^k network – if the image of interest is within the currently displayed network, then it is selected, else the network surrounding the image of interest is shown.
- Temporal browsing – the image of interest becomes the central image. The predecessor and successor images are updated accordingly.
- Historical browsing – the image of interest is added to the list of images visited, so that the user can easily navigate Back to this image if necessary.
- Image/video viewer – this would update to show the high-resolution image of interest.

It is proposed that the temporal browsing pane is shown at all times at the bottom of the window, since this tends to be frequently used. Therefore the image of interest will always be at the centre of this pane. At all times, the interface should be as consistent with current WIMP operating systems as possible. This should enable the user to learn how to use the application quickly and intuitively.

Other required features of the user interface include:

- Progress information – the user should be kept informed as to the current status of the system. For example, a status bar at the bottom of the window could be updated with current information, such as the percentage of search complete.

- Consistency – we have already described our strategy for maintaining consistency across browsing mechanisms. This could be extended by saying that, where the collection and pre-computed data remains the same, all behaviour of the system should be deterministic.
- Cater for different users – the user-interface should be as easy and intuitive to use for inexperienced users, but also flexible enough to cater for the needs of advanced users.
- It should be *screen-aware* and *bandwidth-aware*, as described below.

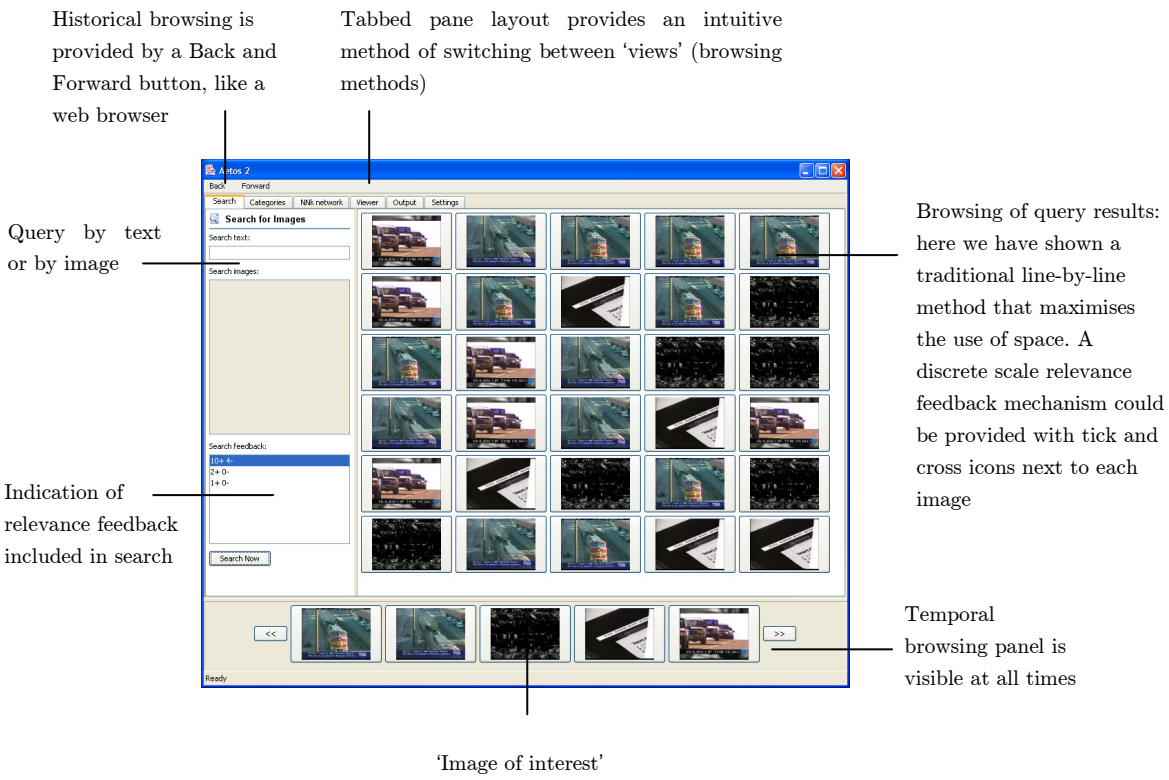
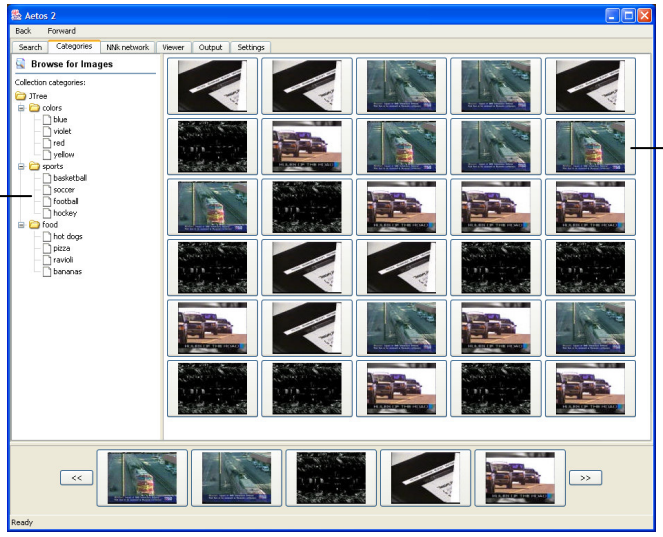


Figure 9. Searching with relevance feedback

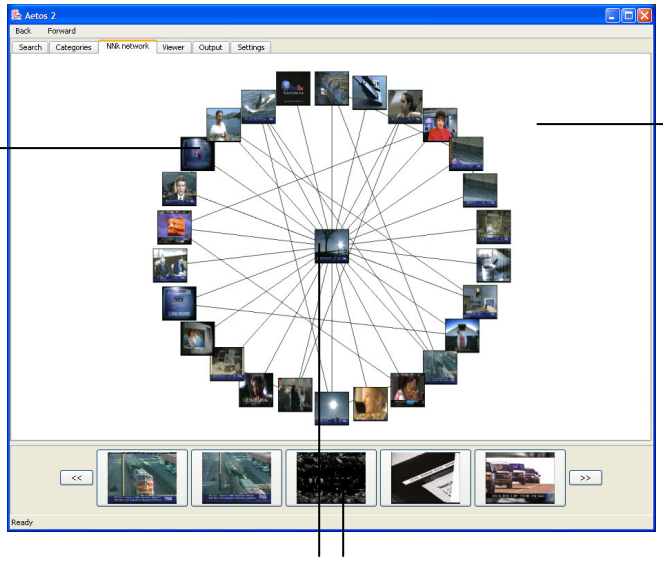
Hierarchical navigation is provided by a tree GUI component showing the categories of the image collection



Browsing of images within a collection: here we have shown a traditional line-by-line method that maximises the use of space. The search function could be used to rank all images within a certain category, so that we could display them in rank order.

Figure 10. Hierarchical browsing

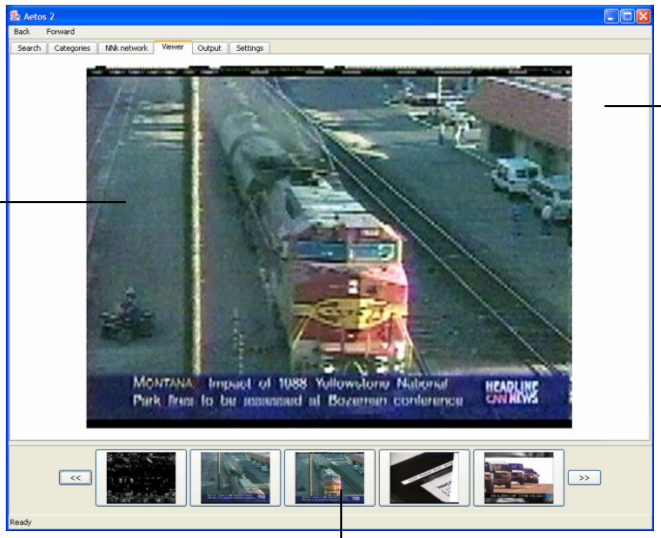
Nearest neighbours are shown in a circle about the 'image of interest'. By drawing the NNk network arcs, we can provide the user with further information as to similarities between images. If the user clicks on an image, that becomes the 'image of interest' and the display redraws to show that image in the centre, surrounded by its nearest neighbours.



'Image of interest'

We could improve use of space by developing a more sophisticated drawing mechanism. We could provide the user with a way to configure the number of neighbours shown, and the depth of network to be shown.

Figure 11. Lateral browsing



An image / video viewer should be included. Here we have shown the high-resolution version of the image of interest.

We could include more features such as a zoom facility or a full screen mode. Also we could display further information about the image, such as EXIF tags or associated annotations.

We could also allow the user to build up a collection of images and browse through their collection in this panel.

'Image of interest'

Figure 12. The viewer

3.4.1 Screen-awareness

The user interface should make appropriate use of the available screen space. That is, it should be able to adapt to both different sizes and different orientations of screen. We shall assume that the minimum screen space available is 800x600. It is uncommon for modern workstations to use a screen resolution of less than this. Ideally, the interface should dynamically update to optimise the layout of components as the user resizes the window.

3.4.2 Bandwidth-awareness

The client should be able to adapt to varying amounts of available bandwidth between the applet and servlet. This might involve decreasing the resolution of the thumbnail images returned if the network connection has limited bandwidth. The client should also make use of caching and pre-fetching techniques to limit the effects of limited bandwidth upon the usability of the browser. If the user does have to wait for data to be transferred from server to client, then they should be informed of its progress by means of a status bar or progress bar.

3.5 CBIR engine (server-side)

The functionality of the server component will be based upon the *Aetos* browser described in [10]. The server component receives search/browse commands from the client and responds accordingly. For example, if the user performs a search using a particular query, the client sends the query to the server, where the search takes place and the images are compared according to their descriptor similarity. The server then responds appropriately with the ranked list of images. Similarly, the user might want to browse the NN^k network. The client sends a command to the server requesting the nearest neighbours of the current image. The server looks this up using the pre-computed index files, and returns the list to the client.

3.6 Improvements

We have determined several areas in which there are scope to improve the level of user-interaction:

- Relevance feedback – further investigation of relevance feedback could be undertaken to try and identify the benefits and drawbacks of a discrete scale feedback system against a continuous scale feedback system. Negative feedback might also be a useful area to explore.
- Poor use of space during retrieval and browsing – it is possible that there are benefits in organising the search results or the NN^k networks to make best use of space. This can expose the user to more results, who is able to quickly and accurately scan the thumbnails for relevant images.

3.7 Extensions

There are many extensions that could be carried out if time permits. These include:

- Search by an external image – currently the user is only allowed to search using an image which has had its feature descriptors pre-computed. This process could be integrated into the browser to allow any image to be used in a query. However, some estimation would be required to normalise the image descriptors in the same way as the rest of the collection.
- Add new image to existing collection – this is similar to the above problem, but has the additional difficulty of adding a new image to the various browsing structures at run-time. It is impractical to re-compute the whole lateral browsing network in order to include the new links to this image.
- Adding annotations to images – this would involve storing some annotation metadata along with each image. It is possible to associate annotations with particular regions of images by adding location information to the metadata.

4 User Interface Design

The user-interface of the browser was developed using a user-oriented approach. At each stage we considered how the features of the interface would enhance the browsing process for the user. Several key concepts of HCI were maintained throughout the design process:

- **Consistency –** This applies both to concepts used within the browser interface and also to concepts used in general in recent human-computer interfaces. For example, if we can bring up a popup menu to apply actions to a particular image in the interface, the user might expect that he could use the same method to bring up a popup menu for any image displayed in the browser, no matter what the current method of browsing. General HCI standards would suggest that the menu should appear when the user right-clicks on an image.
- **Responsiveness –** The user should at no point be presented with an unresponsive interface. This means that there must always be some processing power available to servicing the interface. This is a common oversight in many simple Java programs, in which the event-dispatching thread is used to perform time consuming computation. If the browser is occupied performing computation or retrieval then the user should always have the option to cancel this action. Ideally, the time-consuming task should be performed in the background and the user can proceed with initiating other actions.
- **Informing the user of progress –** When a time-consuming task is taking place, such as computation or retrieval of data from the network we must ensure that the user is kept informed of progress. This serves two purposes: to indicate that progress is still being made (a failure has not occurred) and to allow the user to estimate the time remaining before the time-consuming task terminates.

4.1 Integration of browsing and searching methods

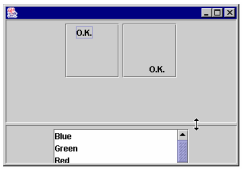
The seamless integration of the different browsing methods is key to providing the user with the benefits of each method during the browsing process. We based the interface design upon the notion of an 'image of interest' as suggested in the prototype interface, as this seems most intuitive to the user. We then considered several paradigms for the integration of the different methods. Firstly, we considered whether the 'image of interest' should be displayed once or multiple times in the interface. The following two options were considered:

- Placing the 'image of interest' in the centre of the screen, with a 'fisheye' visualization of neighbours coming off in horizontal, vertical and diagonal directions to represent browsing in different dimensions. Although this works for a small number of simple browsing methods, such as temporal browsing, it is not conducive to more complex browsing mechanisms. Also it presents each dimension in the same way, so we cannot configure the display depending upon the browsing method used. Separate panels would have to be provided for performing a search or browsing more complex structures such as the tree hierarchy.

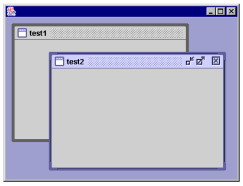
- Splitting the interface into a number of separate panels on the same screen. Each panel presents the functionality of one browsing method. The 'image of interest' is clearly shown in each panel, for example by a coloured border.

It was decided that the second option provided many benefits. By allocating a particular browsing method its own panel, we can ensure the functionality for that method is consolidated in that panel. We can design the presentation of each panel to suit the paradigms of the method being used, whist maintaining consistency with other panels via the 'image of interest'.

Next we considered which GUI mechanism should be used to split the interface into separate panels. We considered the following options:



- Include all panels on the same screen, allowing the user to allocate space by dragging divider bars.



- Separate browsing panels using a desktop-style parent window containing child windows. The user can maximise, minimise, scale and arrange windows as necessary within the parent window.

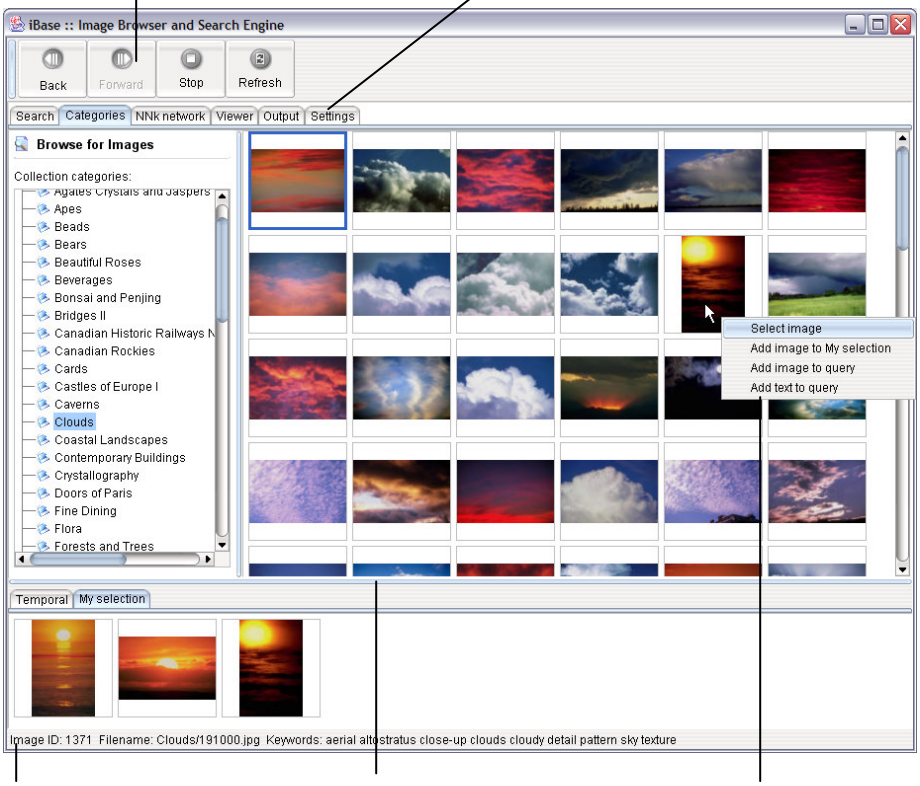
- Separate browsing panels using a tabbed layout. When the user selects a tab for a particular browsing method, that panel is shown and fills the whole screen.

It was decided that the second option was not suitable in this case, since child windows are usually used in cases where the user can open and close a number of documents and apply common actions to them, like in a word processing application. Hence using this model for different browsing methods might go against what the user might intuitively expect.

We decided to use a combination of a tabbed layout and a split pane layout. The final interface layout did not deviate considerably from the prototyped interface. Figure 13 shows the final layout.

Actions that can be applied at any time are located in a toolbar. Most users who are familiar with a web browser will find this interface intuitive.

Tabbed pane layout provides an intuitive method of switching between 'views' (browsing methods)



Statusbar provides user with progress information.

Split pane divider allows user to adjust allocation of screen space between bottom panels and top panels

Context-aware popup menu displays list of possible user interactions

Figure 13. Final interface layout

It was decided that the bottom panel should not only display the temporal browsing panel but also a panel to which a user can add or remove relevant images that he/she finds while browsing. We have called this panel 'My selection'. This helps with the integration of browsing methods since it allows the user to build up a selection of images using one browsing method, and then explore these images further under different browsing methods. Hence the user should be able to have this panel displayed constantly while they change between browsing methods using the upper panel tabs.

To ensure clear consistency between browsing methods we show the current 'image of interest' with a blue border. The user selects an image as 'image of interest' simply by clicking on it. This updates all other views to show this image and hence maintains consistency. We also employ a context-aware right click menu for each image. This presents the user with a pop-up menu containing the set of possible actions for an image in the context of a particular browsing method. This assists with integration since we can use an image from one browsing method in another. For example, if we find a relevant image by lateral browsing, we can then right-click on it and add it to the query images used in a content-based search.

4.2 Search

4.2.1 The searching process

In a traditional text-based search engine, the user inputs one or more keywords which are matched against each image and a ranked list of results are returned. In a content-based search engine, the user inputs a combination of one or more query images (and possible textual keywords also). A content-based search also needs one more ingredient: the feature weights. The weights define how much influence each feature descriptor has on the overall results and can considerably affect the outcome. Figure 14 shows the CBIR searching process which was considered when designing the interface.

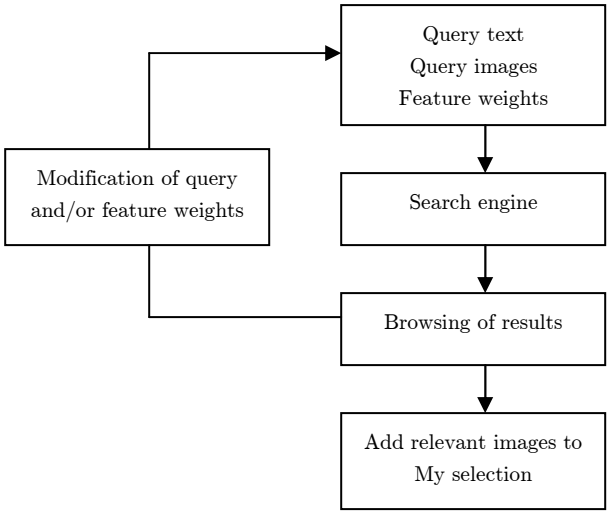


Figure 14. The searching process

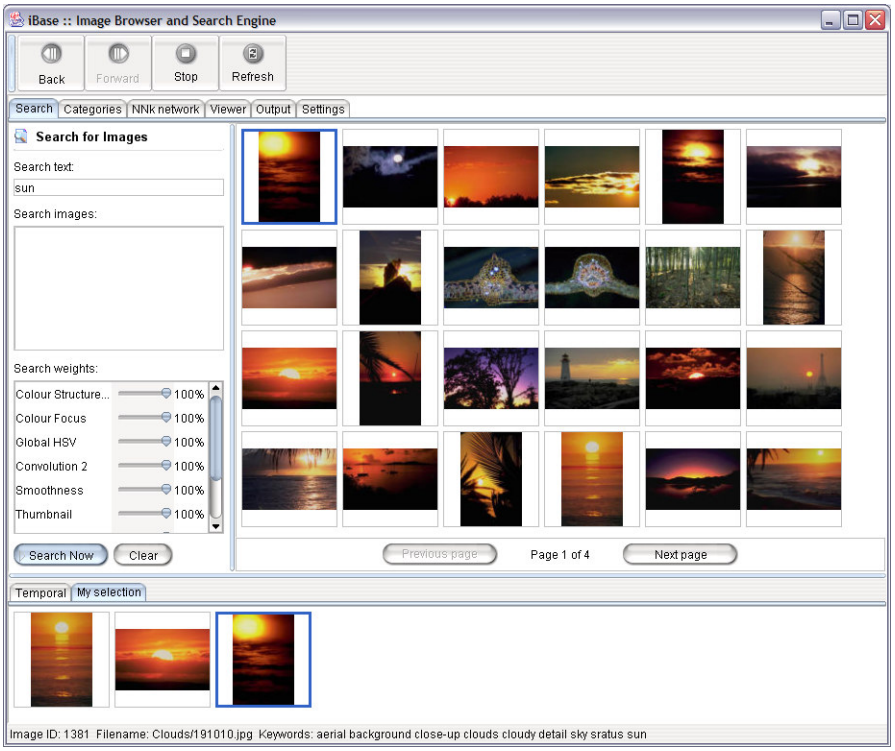


Figure 15. The search interface

Figure 15 shows the final search interface. We have split the panel vertically to give a smaller left-hand panel used for query formulation and a larger right-hand panel used for browsing the search results. This is a commonly used layout for searching mechanisms in operating system GUIs.

4.2.2 Initial query formulation

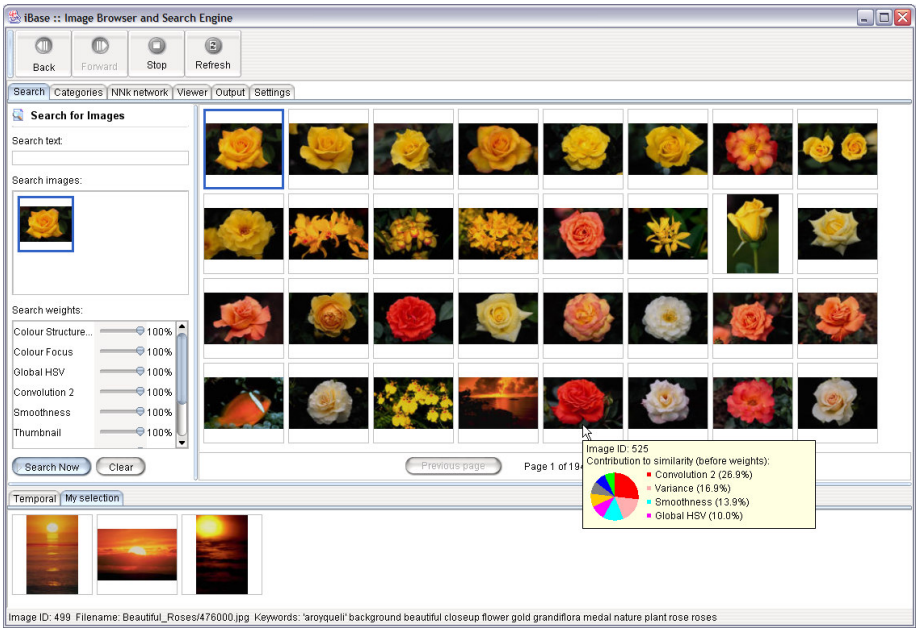
To perform an initial query, the user types text into the 'search text' box and adds one or more image to the 'search images' panel. Any image in the browser can be used in a query by right-clicking on it and selecting 'Add to query'. Often an initial text search is performed to locate a relevant image which is subsequently added to the query. The user can also import an image from an external URL to use in the query. Initially, the feature weights are equally weighted (or as estimated on the sliders by the user), although the user can use weights retrieved from the NNk browser, as discussed in section 4.4.2.

4.2.3 Browsing of results

Results are returned in a page-wise fashion that is common for search engines. This has the advantage that only a limited number of images must be fetched from the server. If the search is clearly not producing relevant results, we do not waste any time or bandwidth retrieving images. The user is able to manually configure how many images are returned on each page, or leave it to the browser to automatically return the number of images that will fit on the screen (screen-awareness). The user browses from page to page with the buttons at the bottom of the results panel.

Results are laid out on each page in a traditional line-by-line fashion, with the most relevant image located in the top-left corner, and the least relevant in the bottom-right corner. Although there has been some research into novel layouts of results, such as clustering or spiralling techniques, we believe that there are advantages in the traditional approach. It is simple and intuitive to the user. We make maximum use of the available space which helps the user to assess the relevance of many images quickly and use them in the search query if required. In this way we are maximising feedback from the user in the searching process.

Because an image is ranked in the results according to the combined distances from the query image(s) under each feature, it is useful to convey to the user the relative contribution of each feature to the total distance from the query image. We considered carefully how to represent this information in a meaningful way. We decided that a pie chart would be suitable as it is only the relative contribution of each feature which is important; the absolute distance to the query image(s) is represented by the position in the results.



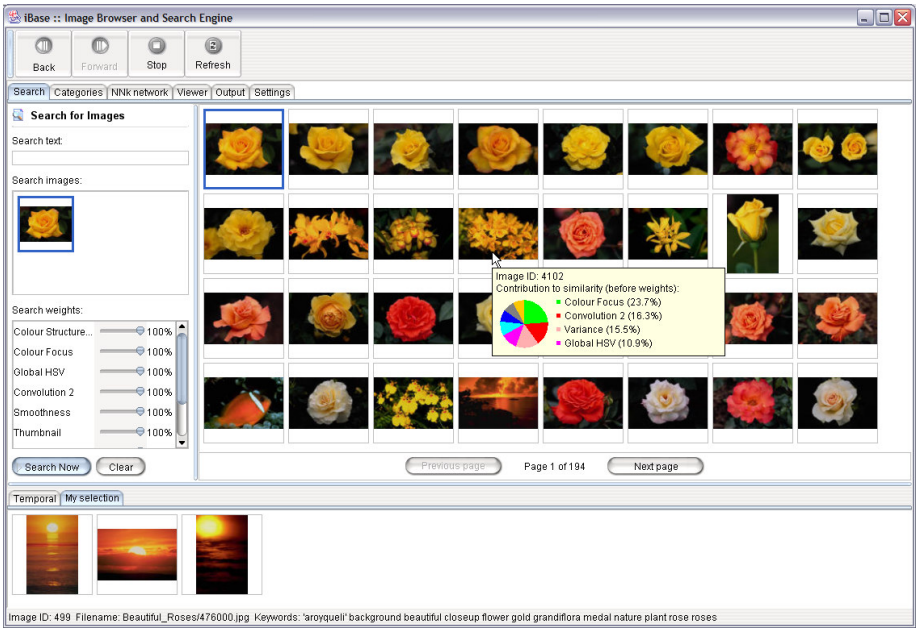


Figure 16 (above) and Figure 17 (below). Explaining the search results

Although the list of images is ranked according to minimum distance (dissimilarity) to the query, it is more intuitive to the user if we display the relative contribution of features in terms of similarity. Initially we calculated for each image the contribution C_f of a feature f to the total similarity as:

$$C_f = 1 - \frac{d_f}{\sum_{i=1}^{n} d_i}$$

where d_f is the distance from the image to the query images and n is the number of features. This ensures that all C_f values are [0,1] bounded. However, after testing it was apparent that this similarity measure does not show relative similarities well when there are many features. Instead we favoured a measure in which we calculate for each image the contribution C_f of a feature f to the total similarity as:

$$C_f = \frac{1}{d_f + \varepsilon}$$

where d_f is the distance from the image to the query images and ε is a constraining factor so that all C_f values are $[0,\ 1/\varepsilon]$ bounded. In practice, we found that a value of $\varepsilon = 0.1$ provides a similarity scale which shows the different feature contributions clearly.

The pie chart is displayed within a tooltip-style box which is displayed after 1 second of hovering the pointer over an image. We display a key to the pie chart for the top four contributors, in order of contribution.
Figure 16-17 demonstrate how this information helps to explain the search results to the user.
Figure 16 shows the pie chart representing the relative contribution of each feature to the total similarity between the yellow rose query image and the yellow flowers that have been returned at position 9 in the results. The pie chart shows that Colour Focus (a colour-based descriptor) is the highest contributors. Figure 17 shows a similar chart for the red rose at position 25 of the results. Here the pie chart shows that Convolution and Thumbnail are the highest contributors, which we would expect since the texture of the rose is very similar to the query image but the colour is different. It is clear that this information would be very useful to the user in the case where it is not immediately apparent which feature descriptors provide the greatest contribution to the similarity rating.

4.2.4 Query modification (relevance feedback)

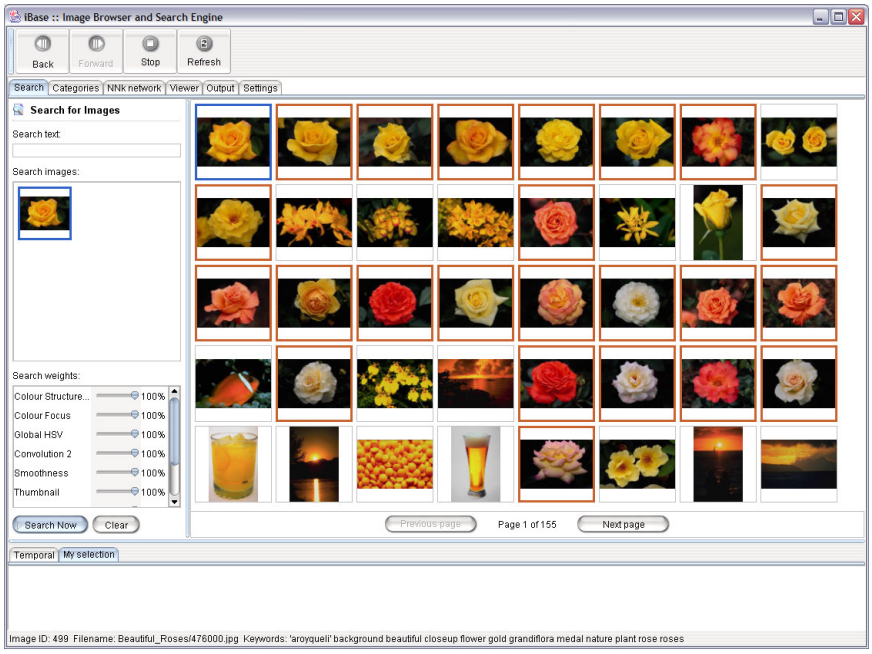
It is difficult for the user to estimate the best combination of feature weights to use for a particular query. We decided to investigate a heuristic approach based upon the similarity contributions computed from the feature distance values, as discussed above. The user adds relevant images to a feedback panel. Weight W_f for a feature f is then computed as:

$$W_f = \frac{\sum_{i=1}^{n} C_f}{n}$$

where C_f is the contribution of a feature f to the total similarity for image i and n is the number of images in the feedback panel. By adjusting the weights in this way, we allocate importance to the features that contributed most to the similarity between the query image(s) and the image(s) chosen as relevant by the user. In this way, we encode information about what the user perceives as a relevant image within the feature weighting. In subsequent searches, images with similar distributions of feature contributions to those chosen by the user as relevant will be ranked higher. After some initial testing, we found that this approach did not provide a substantial gain in performance. Indeed on some occasions it *reduced* the quality of the search results.

We therefore decided to implement a variant of the method discussed in 2.1.4, since this has proved successful in the past. Due to our interface design, we modified the method to use a discrete relevance feedback scale instead of a continuous one. To indicate relevance, the user right-clicks on an image and selects "Mark positive" from the menu, or holds Ctrl and left-clicks on an image. By marking images as relevant, the user indicates this image should have a distance close to zero. The system then computes a new set of weights by minimising the sum of the squared errors between the user allocated distances and the system computed distances as discussed. We evaluate the effectiveness of this method in improving the search performance in section 6.2.
Figure 18 and 19 demonstrate this relevance feedback technique in use.
Figure 18 shows the search result after querying on a yellow rose. The user has selected a number of images as relevant (shown with a red border) and hence assigns a distance close to zero. When the user searches again, the system minimises the error between the user assigned distances and the system computed distances.

Figure 19 shows the updated feature weight sliders and search results using this relevance feedback. We can see that the error is minimised when there is less importance placed on the colour feature descriptors and the results show more roses of different colours. In this way, we are encoding the user's perception of relevance (eg. 'a single rose of any colour') in the feature weight selection.



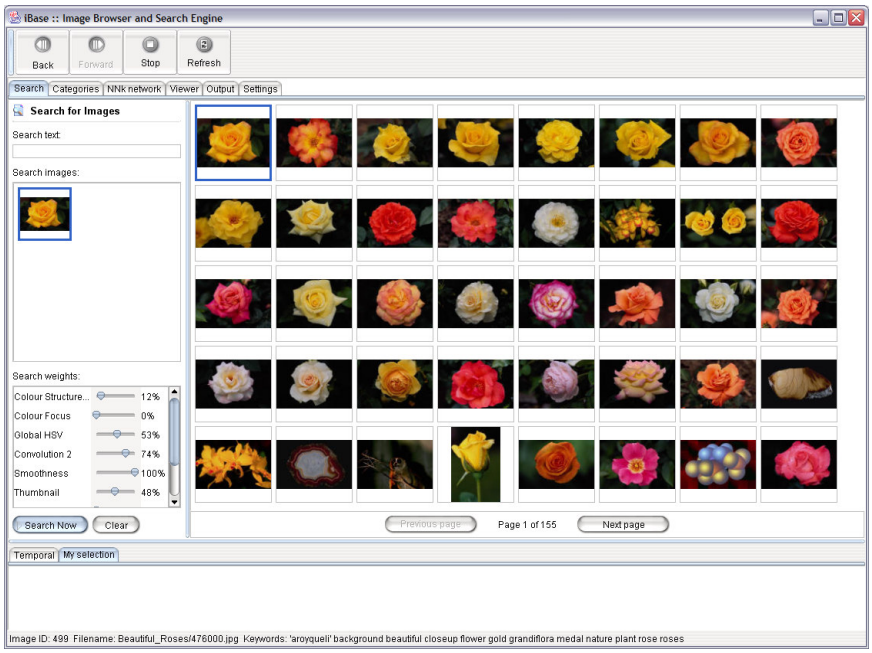


Figure 18 (above) and Figure 19 (below). Relevance feedback

4.3 Hierarchical browsing

Hierarchical browsing allows the user browse for images using a set of pre-defined categories. This lets us take advantage of any expert pre-categorization that has taken place. The hierarchical browsing structure is derived directly from the directory structure on disk. Hence we cater for any arbitrary tree structure allowed by the file system. Figure 20 shows the user interface for the hierarchical browsing panel. Again, we have split the panel into a left-hand panel for category navigation and a right-hand panel to view the contents of the currently selected category. The user can allocate screen space by dragging the divider. We have represented the tree structure as an expandable and collapsible tree interface component. Most users will be familiar with the paradigm, as it is used commonly in operating system GUIs for browsing the file system. The contents of the currently selected category are shown in a line-by-line fashion, in the same way as the search results. This maximises screen space and helps the user to quickly assess whether there are any relevant images. A scrollbar appears if there are too many images in a category to fit on the page. The current 'image of interest' is shown, as usual, with a blue border. If the 'image of interest' is changed in another panel, the panel updates to show the contents of the category in which the 'image of interest' resides. If the user selects a category in the tree which itself contains sub-categories (non-leaf node), then all images in the selected category and all images in each of the sub-categories are displayed.

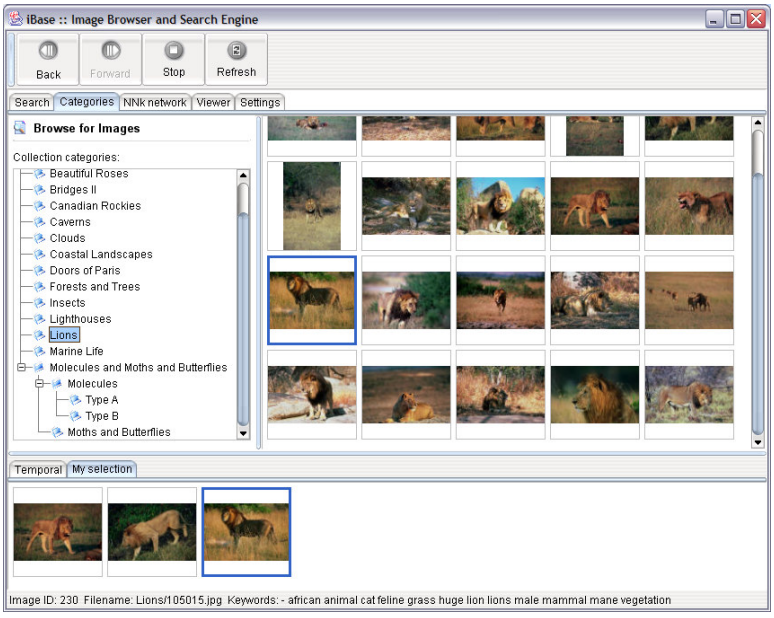


Figure 20. Hierarchical browsing interface

4.4 Lateral browsing (NN^k networks)

Improving the interface and browsing process for the NN^k networks was a particularly challenging area of the project. For each image in a collection we pre-compute the set of other images which were ranked top for some combination of feature weights. In this way, the user can browse across the weight space, and we address the problem of polysemy involved in image retrieval.

We described in 2.3.3 how the network has been presented to the user as a graph where the currently selected image is placed in the centre of the graph, and its neighbours are displayed around it, with the distance to the centre representing the relative similarity. By clicking on a neighbour image, the user can re-centre the graph about that image, and show its neighbours. It is clear that this method has several limitations. This interface does not assist the user in keeping track of which parts of the NN^k network he/she has already explored. The user may find several neighbours that are worth exploring, but then lose track of them while exploring one. Also, it is beneficial if we can modify the network displayed to only show parts of the network that the user has not already explored. In this way, the user does not waste time assessing images he/she has already considered. We also have the problems of overlap and poor use of space associated with a static graph layout.

We have formulated an interface and browsing process which overcomes these problems. Initially we tried displaying the network such that the user could dynamically expand and collapse nodes to view or hide their neighbours. The user could also drag images in the network to arrange them as they wish on the page. This has the problem that the screen would fill up very quickly, and the user would spend much time removing irrelevant images. We modified the strategy so that for each image, the user prunes away irrelevant neighbours (by marking relevant ones) before they are displayed in the network.

Figure 21 shows the final lateral browsing interface. Once again, we have split the panel vertically, allowing the user to allocate space with the divider. On the right-hand side we show the 'image of interest' surrounded by its neighbours according to the NN^k structure. Neighbours are displayed in a square-spiral fashion in order of similarity, to eliminate overlap and maximise use of space. On the left-hand side, the user can select to view either the 'Hub network' (static network showing 36 most highly-connected nodes) or 'My network' (dynamic network in which the user can add relevant nodes in order to systematically explore the neighbours of each). The hub network can provide a useful entry point for browsing the network.

4.4.1 The browsing process

Below we describe each step involved in browsing the NN^k network for relevant images using our interface. We use an example topic from the TRECVID 2003 search task. The problem is to find all images showing an aeroplane appearing to take off.

1. Select an initial 'image of interest'

Figure 21 shows the initial browsing interface after an initial image has been selected. This can either be from the hub network or from elsewhere in the browser, such as the search results. In the example we have performed a text-search for 'plane' to locate the selected image. The top 24 neighbours of this image in the NN^k network are then displayed.

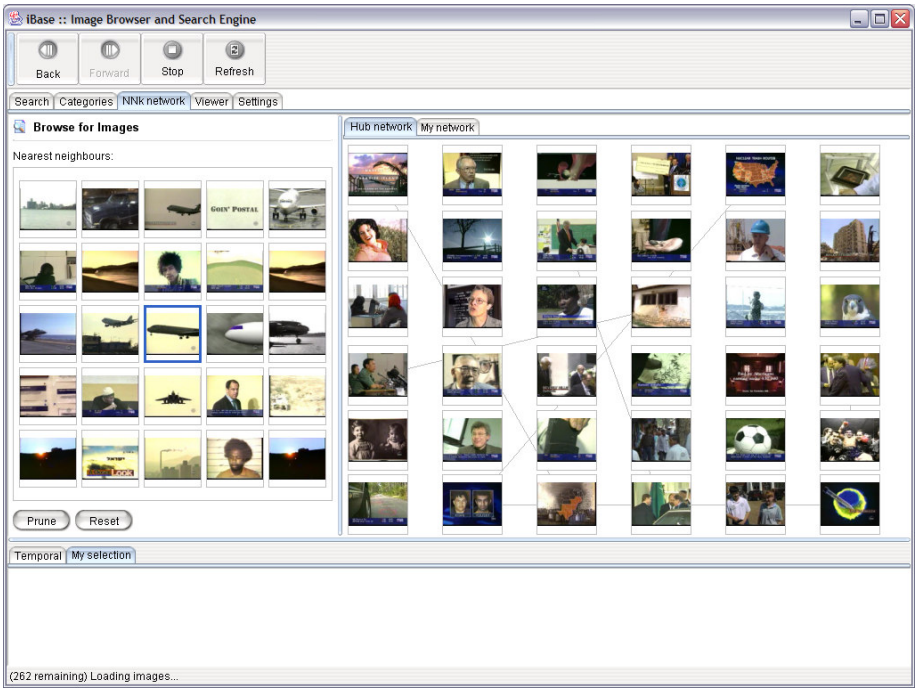


Figure 21. Lateral browsing interface after initial image selection

2. Assess neighbours for relevance

We can observe that there are several relevant images (i.e. planes taking off). There are also several images which are not directly relevant, but may lead to relevant images in a different area of the NN^k network (e.g. planes on the runway) and so should not be immediately dismissed. The rest of the images are irrelevant and can be disregarded for the duration of the browsing process. By selecting 'mark positive' from an image's right-click menu, or holding Ctrl and left-clicking on an image, the user can declare to the system that that image is relevant (or is not directly relevant but should be marked for further exploration). Figure 22 shows the positively marked images drawn with a red border.

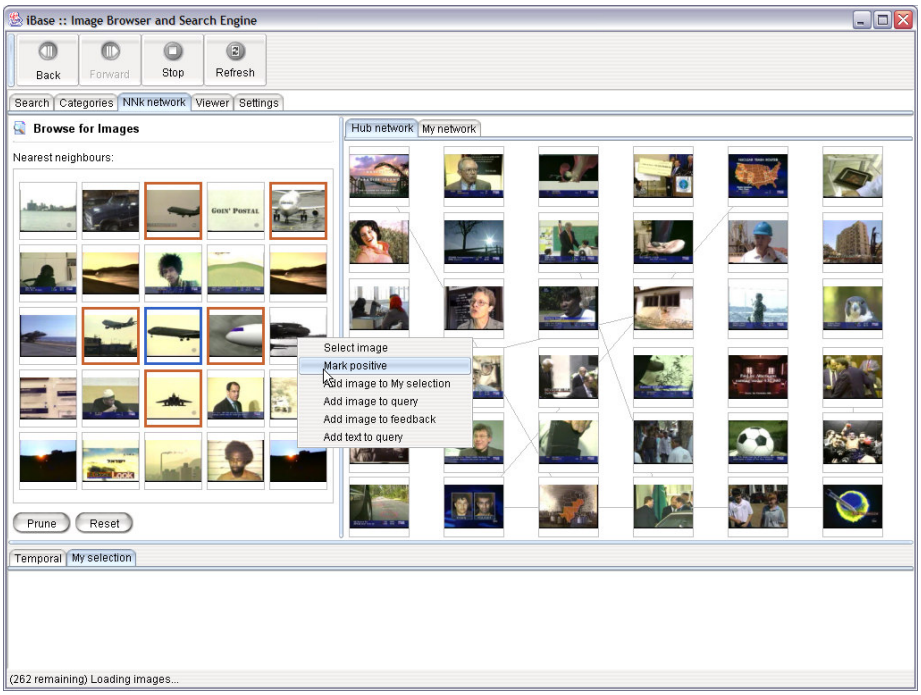


Figure 22. Lateral browsing interface after relevance feedback

3. Prune irrelevant neighbours and add relevant neighbours to user network

Having marked the relevant images, the user can prune away the irrelevant images by pressing the prune button. This removes irrelevant images from the display and leaves only relevant images. The irrelevant images will not appear as a neighbour of any other image in the collection, since the user has already assessed them as irrelevant. The user can undo this action by pressing the reset button.

Pressing the prune button also adds the selected image and its pruned neighbours to the 'My network' panel. Here the neighbours are shown in a network where an arc between images A and B represents the fact that image A is a neighbour of image B or image B is a neighbour of image A. The user can arrange the network as they wish by dragging the images.

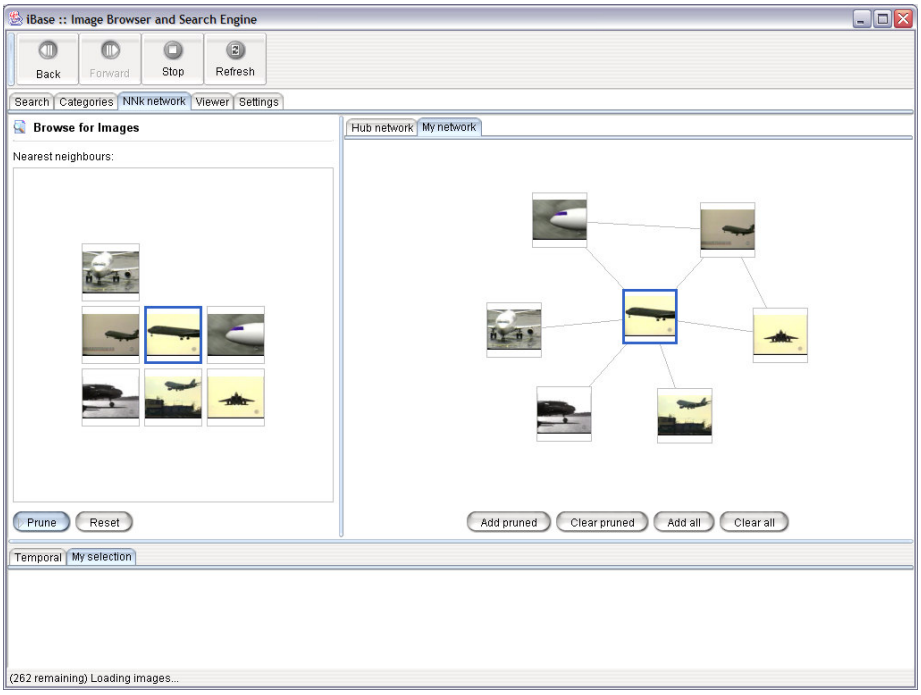


Figure 23. Adding relevant images to the user network

4. Repeat steps 2 and 3 for all un-pruned images in the network

Having added relevant images to the network we can now systematically apply the pruning steps to all un-pruned images in the network. Figure 24 shows the effect of pruning one of the neighbours of the initial image. The network grows and we can now in turn explore this image's relevant neighbours. To help the user keep track of the parts of the network already explored, we show pruned images with a black border. All neighbours that have already been assessed as relevant or irrelevant are not shown in the neighbours of un-pruned images, until the user presses 'Reset', to reset the pruning decisions of one image, or 'Clear All', to reset the whole browsing process and clear the user network.

The user can add images from the 'My network' panel to the 'My selection' panel at any time by pressing 'Add pruned' or 'Add all' to add only pruned images or all images respectively. To prevent the panel from becoming cluttered, the user can press 'Add pruned' and then 'Clear pruned' to remove pruned images from the network, and then continue exploring un-pruned images.

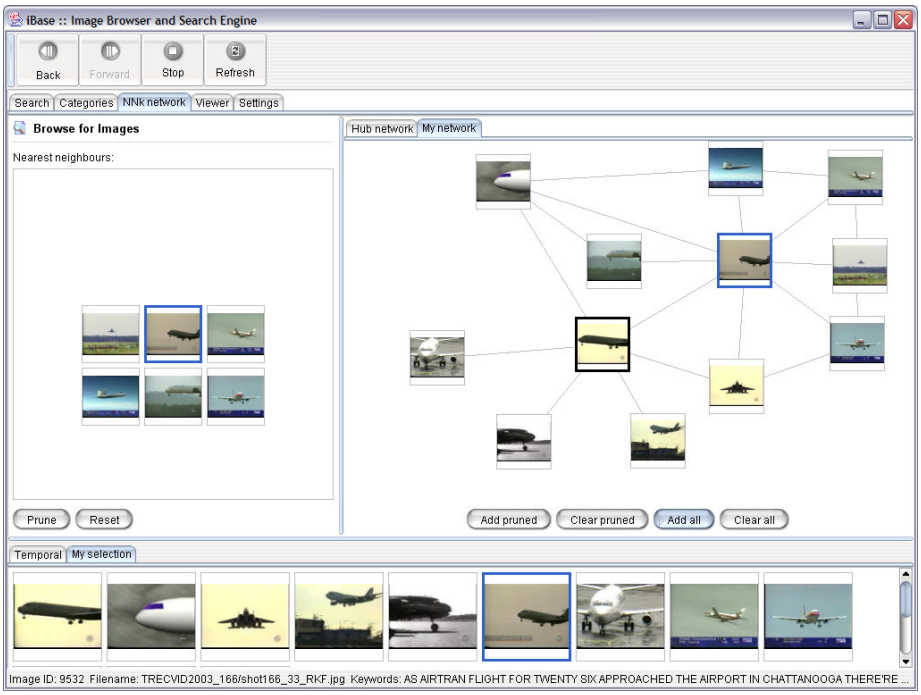


Figure 24. Exploring the network

Using this technique we can very quickly build up a large collection of relevant images for complex queries which are difficult to describe in a search, possibly due to the high-level nature of the topic or the difficulty in feature weight estimation.

4.4.2 Explaining neighbour images

In our discussion of the browsing of search results, we demonstrated how we could provide the user with a pie chart showing the contribution of each feature to the total similiarity of an image compared to the query image(s). This helps to explain why a particular image appears in the results. We can employ a similar technique when browsing the neighbours of an image in the NN^k network. However, the problem in this case is more difficult, since we need to convey weight information also.

Recall that an image I is a neighbour of an image Q if there is a feature vector for which I is ranked top under the combination of all feature discriptors describing the similairty between I and Q. Hence one neighbour corresponds to the top image for some region in n-dimensional feature space, where n is the number of features. This is a very difficult concept to visualize and therefore difficult to convey to the user in a meaningful way.

We must show for each image not only the contribution of each feature to the total similiarity of a neighbour but also the relevent region in weight-space which adjusts the contribution of each feature. In order to retain simplicity and consistency, we decided to take the centroid of the area in weight-space represented by a neighbour, and hence show the weighted contribution C_f of a feature f as:

$$C_f = \frac{w_f}{d_f + \varepsilon}$$

where d_f is the distance from the image to the central image, w_f is the weight of feature f at the centroid of the region in weight-space for which this image was ranked top, and ε is a constraining factor so that all C_f values are $[0, 1/\varepsilon]$ bounded.

We draw the relative contributions of each feature on a pie chart as before. Figure 25 shows us that the image to the bottom-left of the central image is a neighbour because it was ranked top under a feature regime which places heavy importance on the Thumbnail descriptor.

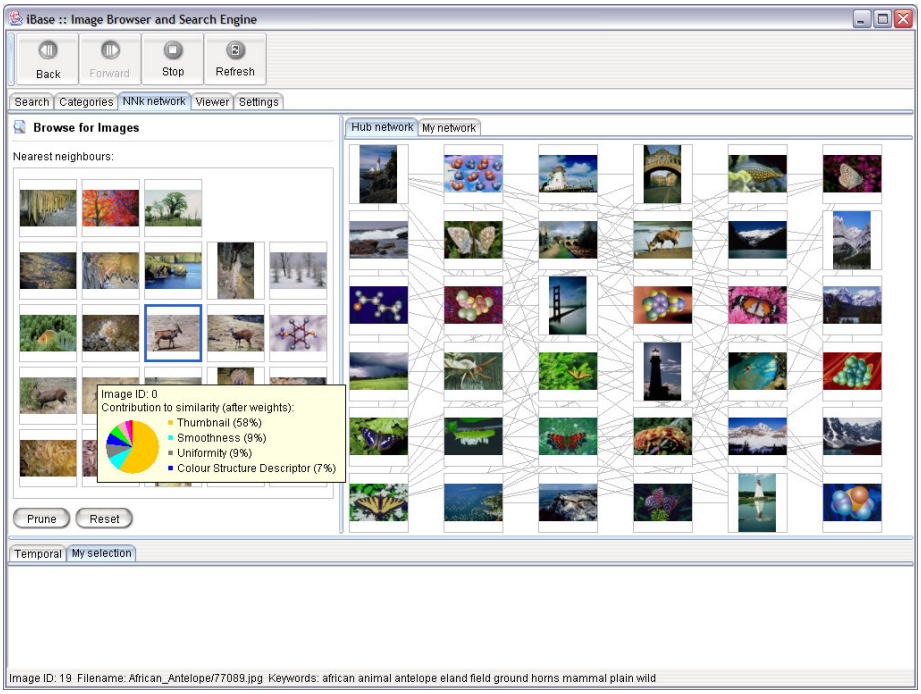


Figure 25. Explaining neighbours

By right-clicking on a neighbour, a user can add that image to the feedback for a content-based search. This updates the weights sliders according to that neighbour's weight-space centroid. This follows the same principle as relevance feedback described in 4.2.4. By indicating to the system which features are important for describing similarity to the query image under the user's perception of relevance, we can improve the performance of the search. Search results will favour those images which are similar under that weighting of features.

4.5 Temporal browsing

In an image collection, we might have temporal information which tells us the predecessor and a successor of each image. For the TRECVID collection of video shots, we can determine each image's temporal neighbours from the video sequence. The temporal neighbours of a relevant image may also be relevant since the shots are adjacent in the video sequence. Figure 26 demonstrates temporal browsing during a search for images of Osama Bin Laden. The temporal browsing user interface is at the bottom of the screen.

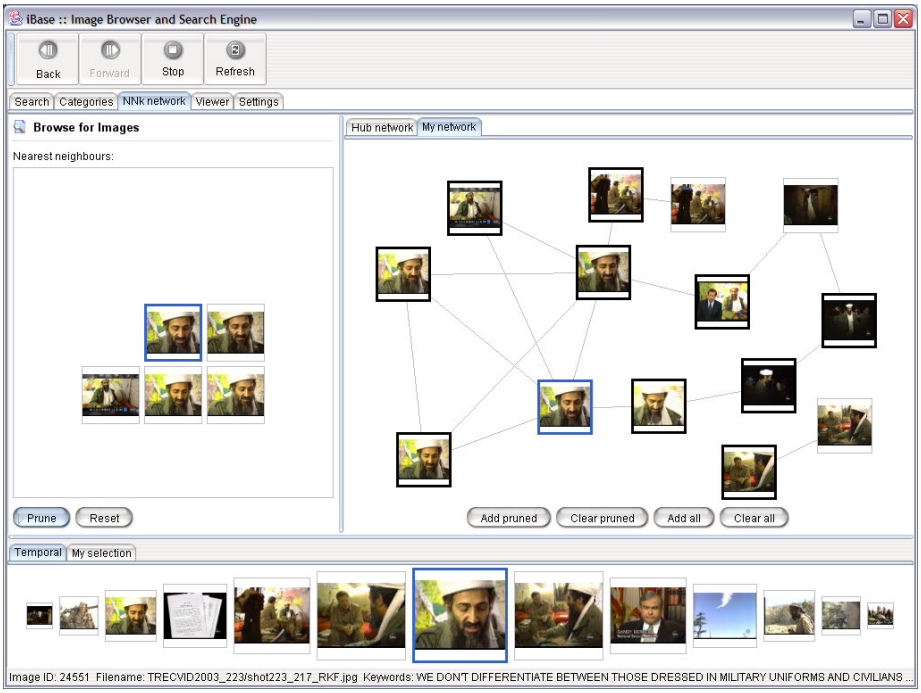


Figure 26. The temporal browsing interface

The 'image of interest' is shown in the centre of the screen. The user can then look forwards and backwards along the sequence of images in the temporal dimension. The size of each shot decreases with distance from the image of interest. This maximises use of screen space and reflects the fact that images closer to the centre are likely to be more relevant than those further away.

4.6 Historical browsing

The ability to retrace one's steps is paramount in any browser. This functionality has become commonplace in applications such as file system browsers and web browsers, to the extent that this functionality is expected by the user.

We must be careful to define exactly what is expected of historical browsing capabilities, and to what extent it should recreate the prior state of a system when the user decides to go 'back'. There is a clear distinction between recreating the prior system state in terms of *navigation* and in terms of *user action*. The paradigm that we shall adopt is that prevalent in HCI today. *Navigational* historical browsing is provided by 'back' and 'forward' buttons.

This allows the user to go backwards and forwards along the past navigation path of objects. In the case of a file system browser, this is the path of directories viewed and in the case of a web browser this is the path of web pages viewed. In our case it is the path of images viewed. In contrast, historical browsing of *user action* is provided by 'undo' and 'redo' buttons. This allows the user to go backwards and forwards along the past sequence of user actions. For example, in a file system browser, the user may be able to undo a file copy operation by pressing the 'undo' button. Pressing the 'back' button would just change the view to the previous directory, leaving the file copy operation intact.

Under this interpretation, we decided to provide historical browsing for navigation but not for user action. This is because most user actions in our system (eg. 'Add image to My Selection') are very simple to undo with the provided inverse options (eg. 'Remove image from My Selection'). Preserving the exact state of a complex system every time the user performs an operation is also a time-consuming implementation task that adds little power to the browsing process. In contrast, navigational historical browsing can be implemented relatively easily and adds important functionality to the browser.

We decided that browser should maintain a list of the past 'images of interest' so the user can retrace their steps and each panel would update accordingly to show this image under that view. There is then the problem of how the search panel should update when the past 'image of interest' resides in the search results of a prior search. It was decided that we should also preserve the query parameters of each search, so that when we go back through the path of selected images the search results at that time are recreated.

4.7 Image viewer

The image viewer provides the user with the full-resolution version of the current 'image of interest'. It also shows the full details (ID, filename, resolution and associated caption) of this image. This is particularly useful to browse the collection of images that the user has built up in the 'My Selection' panel. Figure 27 shows a case in which the user is unsure from the thumbnail whether the image does actually show Bin Laden. The high-resolution image confirms that the figure being interviewed is in fact Bin Laden.

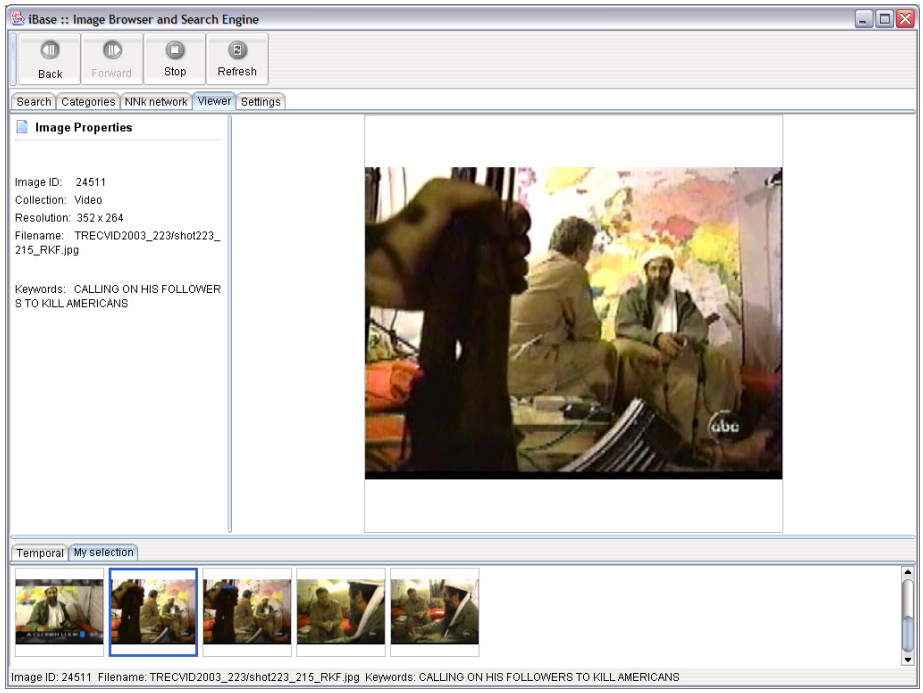


Figure 27. The image viewer

4.8 Settings

Figure 28 shows the settings panel from which the user can configure the browser. On startup, the collection listbox is populated with image collections served by the server. The user can switch between collections simply by selecting it in the listbox and pressing 'Apply'. When the user switches to a particular collection the feature listbox is populated with the implemented feature descriptors for that collection. By default all features are selected. The user can alter the combination of features used by the image search by selecting them and pressing 'Apply'. The user can also configure many other program settings, by selecting the relevant setting in the 'program settings' listbox, altering the value in the textbox and pressing 'Apply'. This maximises the flexibility of the browser and ensures the user is restricted as little as possible by minor design decisions, such as thumbnail display size. In the right-hand panel, we log all communication with the server and also any exceptions that may occur.

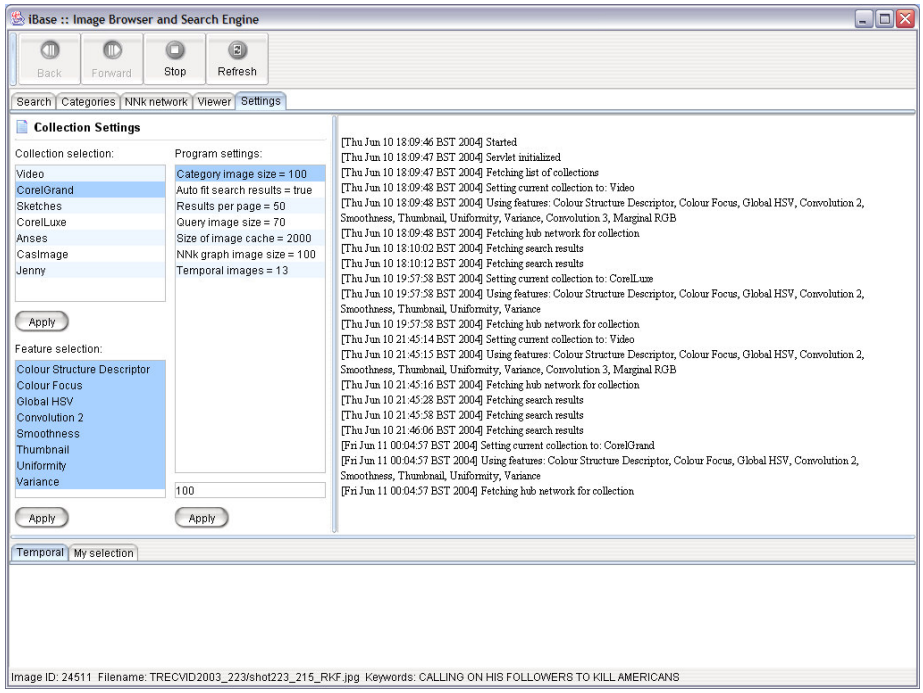


Figure 28. The settings interface

5 Implementation

5.1 System architecture

It was decided at an early stage that the implementation should be web-based. This provides platform independence on the client-side and hence wide accessibility, allowing many users to browse a collection simultaneously using a single server instance. Users only need a standard web browser and do not need any additional software to run the browser. However, it was accepted that a user may be required to install one of the common web-browser plug-ins to enable interactive content within web pages. We discuss the various options below.

5.1.1 Implementation language

5.1.1.1 Server component

Static HTML generation

The simplest web-based image browsers take an image collection and compute a set of static HTML pages for every possible state that the browser could be in. In this way, the user simply navigates between the generated web pages to browse through the collection. This model is fine if the collection is small and the browsing method is simple (eg. hierarchical). However, our browser should cater for very large image collections and a whole range of browsing methods. Hence it is not practical and almost impossible to compute static HTML pages for every single state of the browser. Obviously where searching is involved we will need an interface that dynamically updates according to the search results. Therefore the concept of static HTML generation was dropped at an early stage in the project.

Dynamic HTML generation (JSP / PHP / ASP)

More sophisticated web-based applications can be produced using server-side scripting to dynamically generate pages to pass to a web browser or client. This method is usually used to directly generate a pure HTML interface. Once a page has loaded, it is relatively static and only a limited amount of user interaction can take place on the client-side before another page must be retrieved from the server. Data can be passed (eg. using XML) to other client software such as a Java applet or Flash presentation, but this incurs encoding and decoding overheads.

Java servlet

Java servlets are Java programs that inherit from the `HttpServlet` class and override the `doGet` (or `doPost`) method. They are run inside an application server environment (such as Apache Tomcat) which passes on a web request to the servlet by triggering the `doGet` (or `doPost`) method of the servlet. The servlet then processes the command and provide a stream of dynamic HTML or other output. The advantage of using Java servlets is that it is easy and efficient to send Java objects from the server-side process to a client-side Java applet and vice versa. Most of Java's data types are serializable and hence any custom object composed purely of such objects is also serializable. This means that little additional marshalling code is required from the programmer, and we can use the standard HTTP protocol.

Others (eg. C++/.NET)

There are many other options for implementing the server-side process, including C++ and .NET. Although we could gain efficiency in server-side processing, these would be standalone server processes and as such the protocol for communication with the client component would have to be defined at a much lower level. This would require a great deal more work and might take time away from the main issues in the project.

It was concluded that a Java servlet would provide the most effective server-side solution, since Java is a powerful language with a wide library of classes. By using a servlet, we can perform easy communication to an applet client with minimum overheads and additional work for the programmer.

5.1.2 Client component

We narrowed the implementation options for the client component to a pure HTML interface, an embedded Java applet, or Flash content. We also considered a mixture of these technologies in order to take advantage of the strengths of each. We summarise the advantages and disadvantages of each in turn below.

HTML

Advantages	Disadvantages
• Maximum compatibility with client platform – only need a standard web browser • Javascript can be used for (limited) client side processing • Simplicity	• Restricted to web page style interaction • Restricted client-side processing • Poor client context awareness – difficult to adjust for screen size and bandwidth capacity

Overall it was felt that a pure HTML interface was too restrictive in client-side operations. Since the project's focus is on reaping the benefits of a seamlessly integrated interface and increasing the level of user interaction, a more powerful client side technology was thought to give more scope in providing a rich set of user interactions which would assist the searching and browsing process.

Macromedia Flash

Advantages	Disadvantages
• Rich user interaction possible • The *defacto* standard for multimedia presentation on the internet • Plug-ins available for numerous browsers/platforms	• Not suited to web application client-server communication, more appropriate for client-side presentations and animations • Non-standard and limited ActionScript programming language • Supporting browser / plug-in required

Although use of Flash has become widespread on the web in recent times, it is more suited to presentation of vector graphics and animations. A high level of user interaction can be achieved but it is uncommon that it is used as the front-end for a web-based application where there is much communication with a backend server. Graphics must be designed and imported by the user, and there are limited pre-assembled interface components. Client-side processing can be performed in ActionScript. This is Macromedia's own language which was developed simply for performing client side operations in Flash presentations. It is by no means as comprehensive or as powerful as mature languages such as Java.

Java Applet

Advantages	Disadvantages
• Extensive libraries of interface components (AWT/Swing) • Can harness the full power of Java to provide maximum flexibility in developing client side functionality • Tried and tested applet-servlet communication can be used • Plug-ins available for numerous browsers/platforms	• Plug-in is reasonably heavyweight • Supporting browser / plug-in required

Java applets have become another method of delivering feature-rich applications on the Internet. They allow a user to run a Java program inside a web page as if it were a regular Java application. Of course because Java Applet code is downloaded and executed on the fly from a website, there are extra security implications for the client machine. It is usual for an applet to run in a restricted 'sandbox' environment, with limited privileges on the local machine. Additional permissions, such as access to the local file system, can be granted if the applet is signed. In this case, the user is presented with an applet's digital certificate before executing any code and the user must confirm their trust for the source.

It was concluded that a Java applet implementation has the power and flexibility to provide the rich set of user interactions that will bring real benefits to the browsing and searching process. Most major platforms and browsers support the Sun Java plug-in.

Swing vs AWT

Traditionally, applets have used the Abstract Window Toolkit (AWT) set of classes to provide basic GUI features which are platform independent. However, the newer Swing library provides many advantages over AWT. We discuss the main differences below:

- AWT components hide differences between GUI components on different platforms by taking the least common denominator approach. This restricted AWT components to the functionality present on every supported platform.

- Swing components are implemented with absolutely no native code. Hence they are not restricted to features that are present on every platform and can have more functionality than AWT components.

- Swing components extend AWT capabilities. Examples include:
    - Displaying images in buttons and labels
    - Extensive control of component borders
    - Support for multiple look-and-feels
    - Components do not have to be rectangular

The following excerpt was taken from Sun's website:

"Although the Java 2 Platform still supports the AWT components, we strongly encourage you to use Swing components instead▷"

Hence we shall develop our user interface using Swing components. The only apparent disadvantage is that client machines will need at least Sun Java plug-in 1.1.1.

5.1.3 Architecture and component interaction

The focus on a web-based system led to the development of a three-tier architecture found in many web applications. In this model, multiple clients can connect to a single server process which retrieves data from the image collection and index files, performs computation and returns the results to the client. The idea is that the bulk of the processing is taken away from the client and we ensure that the underlying data is only accessed by the centralised (and trusted) middleware.

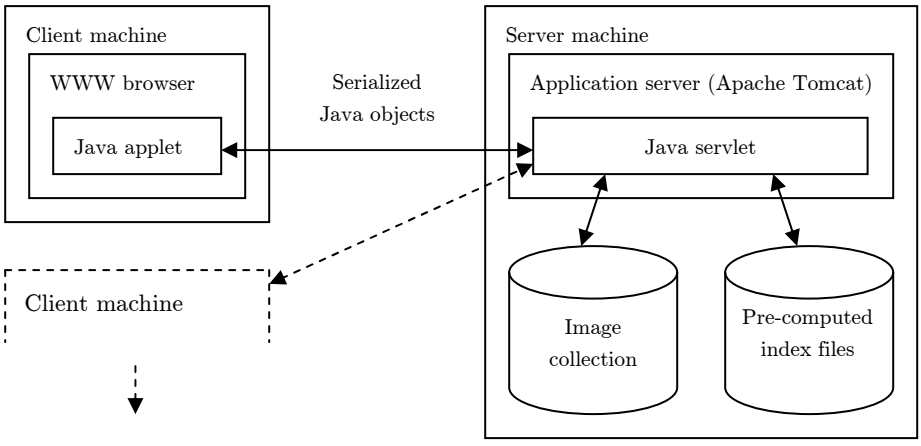


Fig 12. Final system architecture

5.1.4 Image collection and index files

An image collection is stored on disk in a hierarchical structure, either in human-allocated categories or arbitrary directories. Alongside the image data itself are various pre-computed index files which reduce the need for runtime computation. It is these index files that store the relationships between images that we will be exploring in the browser, and also feature descriptor values for images to facilitate a searching mechanism. Figure 13 shows the file structure used for image collections. The purpose of each part is described below.

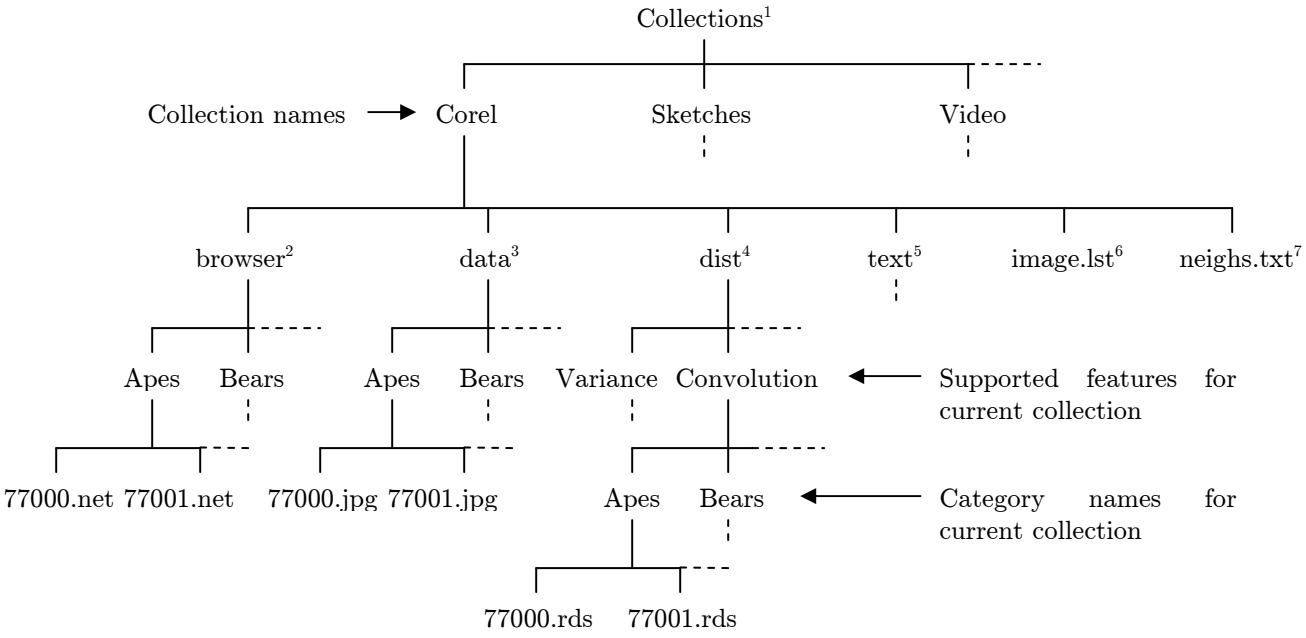


Fig 13. File structure of image collections

1. All image collections are stored under one directory. The servlet process is initialised with this location on disk.

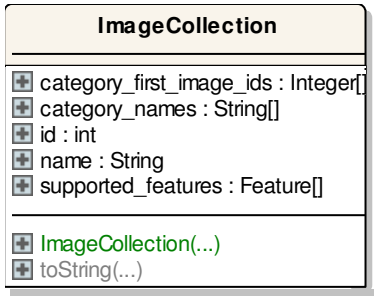
For each collection:

2. The **browser** directory contains index files which contain a list of the IDs of the NN^k neighbours for each image, and the weight centroid values for which this image was ranked top by the combined feature distances. These files are in the same hierarchical structure as the image data itself.

3. The **data** directory contains the actual image files preserved in the hierarchical structure from which the hierarchical browsing structure is directly derived.
4. The **dist** directory contains directories for each of the supported features in the current collection (different feature sets are suited to different types of image collection). For a particular feature, we store an .rds file for each image which contains distance values to all other images in the collection (truncated to the top 1000 shortest distances). This can be used in calculating content-based search results.
5. The **text** directory contains index files holding the text annotations/keywords for each image. This is used to implement a text-based search.
6. The **image.lst** file is a list of the filename and path of every image in the collection. The position in this file is taken as a particular image's unique ID.
7. The **neighs.txt** file is a list of the image IDs for the left and right temporal neighbour of each image in the collection. This is generated from the position of an image in a video sequence or from timestamp information. If there is no temporal information, we simply allocate the left and right temporal neighbours to be the previous and next images as ordered by the file system.

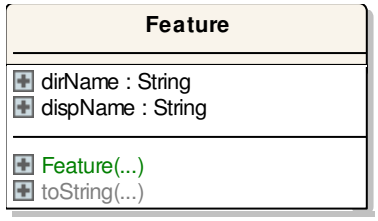
5.1.5 Common data types

The requirements of each method of browsing and searching an image database were taken into consideration when formalising the data types used in the system. The system is based around several key user-defined data types (Java classes):

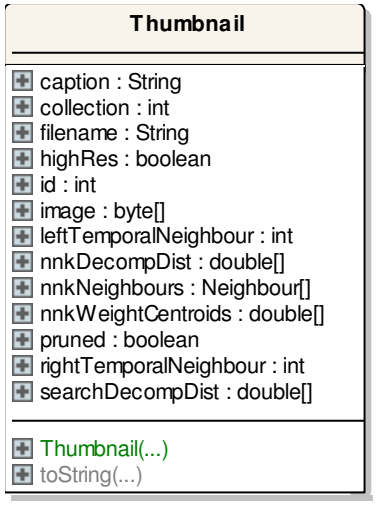


The `ImageCollection` class holds data associated with a single image collection. Required parameters include:

- Image collection ID, as allocated by the servlet
- Image collection name (directory name on disk)
- List of features supported by this collection as `Feature` records (see below)
- List of category names in this collection (as given by hierarchical file structure)
- List of image IDs corresponding to the first image in each of the categories.



The `Feature` class holds the directory name and corresponding display name for a given Feature.



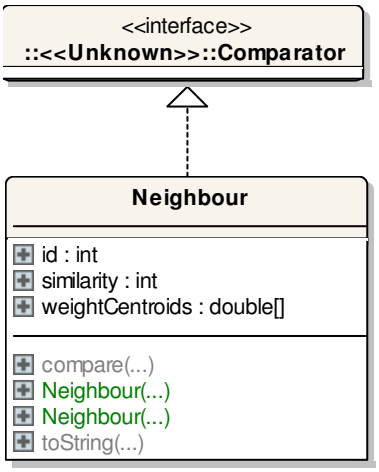
The `Thumbnail` class holds all data associated with one image in a collection as well as the actual image data itself. Required parameters include:

- Collection ID and Image ID to uniquely identify an instance
- Filename of the image as in the hierarchical file system
- Associated caption if available
- Image itself (byte array)
- IDs of left and right temporal neighbours
- IDs of neighbours in NN^k network
- Flag stating whether the thumbnail contains the high resolution or low resolution version of an image

Optional parameters include:

- Distances (under each feature) to the currently selected image and centroids of weights for which this image was ranked top (used in presenting the NN^k network, as described in 4.4.2)
- Distances (under each feature) to the current query image or set of query images (used in presenting the search results, as described in 4.2.3)
- Flag stating whether the current image has been pruned in the NN^k browsing process or not (used during browsing process of the NN^k network, as described in 4.4.1)

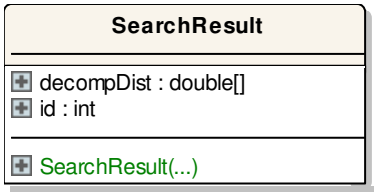
Optional parameters are populated as needed by the client applet, for example, when search results are retrieved.



The `Thumbnail` class has an association relationship with the `Neighbour` class. This contains information about an image's neighbour in the NNk network:

- The image ID of the neighbour
- The similarity of the neighbour to the image, represented by the proportion of the weight space matrix for which this image was ranked top.
- The centroid of the area in weight space for which this image was ranked top.

`Neighbour` implements the `Comparator` interface, so that we can sort an array of `Neighbours` by similarity for use when presenting the NN^k network to the user.



The `SearchResult` class contains information about an individual result when returned as part of the search results. This includes the image ID and distances to the query image(s) under each active feature.

5.1.6 Server-client interface

The server-client interface was designed with simplicity in mind to make it easy to use the same client-side interface with a different server-side implementation. The interface is implemented using standard applet-servlet communication. We package requests and replies (with relevant parameters) into a Java `Vector` object which are serialized and sent across the network as HTTP. Figure 29 shows possible client queries and corresponding server replies.

Query	Parameters	Response
`getCollections`	`N/A`	`ImageCollection[] collections`
This query returns an array of `ImageCollection`s for use when initialising the client applet		
`getImageByID`	`int collectionID` `int imageID`	`Thumbnail image`
Given a collection ID and image ID, this query returns the relevant `Thumbnail` from the image collection with low-resolution image data.		
`getHighResImageByID`	`int collectionID` `int imageID`	`Thumbnail image`
Given a collection ID and image ID, this query returns the relevant `Thumbnail` from the image collection with high-resolution image data.		

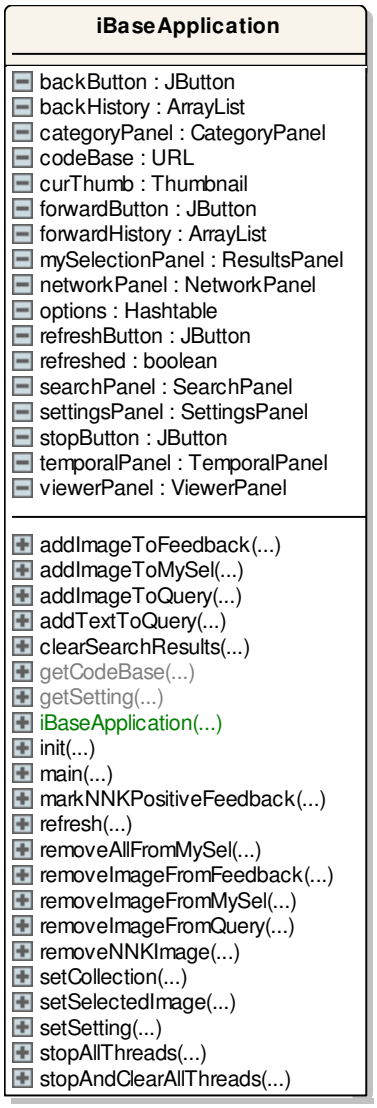
`getSearchResults`	`int collectionID`
	`int[] queryImageIDs`
	`String queryText`
	`boolean[] activeFeatures`
	`double[] featureWeights`
	`int page`
	`int resultsPerPage`
		`SearchResult[] results`

This query is used to fetch search results from the server. The client must pass the query image IDs and search text, as well as details of which features are active and their relative weights. The server uses these parameters to return an array of `SearchResult`s, ranked from most similar to least similar for the specified page.		

Figure 29. Client queries and server responses

5.2 The applet

5.2.1 Client architecture



We implemented the client interface using Java's Swing library of interface components. The main window is constructed as a `JFrame`, from which we inherit in the `iBaseApplication` class. This class is responsible for constructing the overall layout of the interface and the various different browsing panels. It also handles all communication between different parts of the interface. It is this mediator design pattern that is responsible for providing the integration between different browsing methods in the applet. Each time the user selects a new 'image of interest' in one panel, a call to `setSelectedImage()` on `iBaseApplication` cascades this selection to all other browser panels.

The panels for each method of browsing are implemented as classes which inherit from `JPanel`. Each panel is responsible for constructing its own interface and implementing the functionality specific to that method of browsing. To change the status of another panel, a class may call the relevant method on the `iBaseApplication` class, which will pass the request to the appropriate panel.

Figure 30 shows the inheritance and association relationships between classes in a UML diagram. We have only shown the main components of the interface to avoid the diagram becoming too complex.

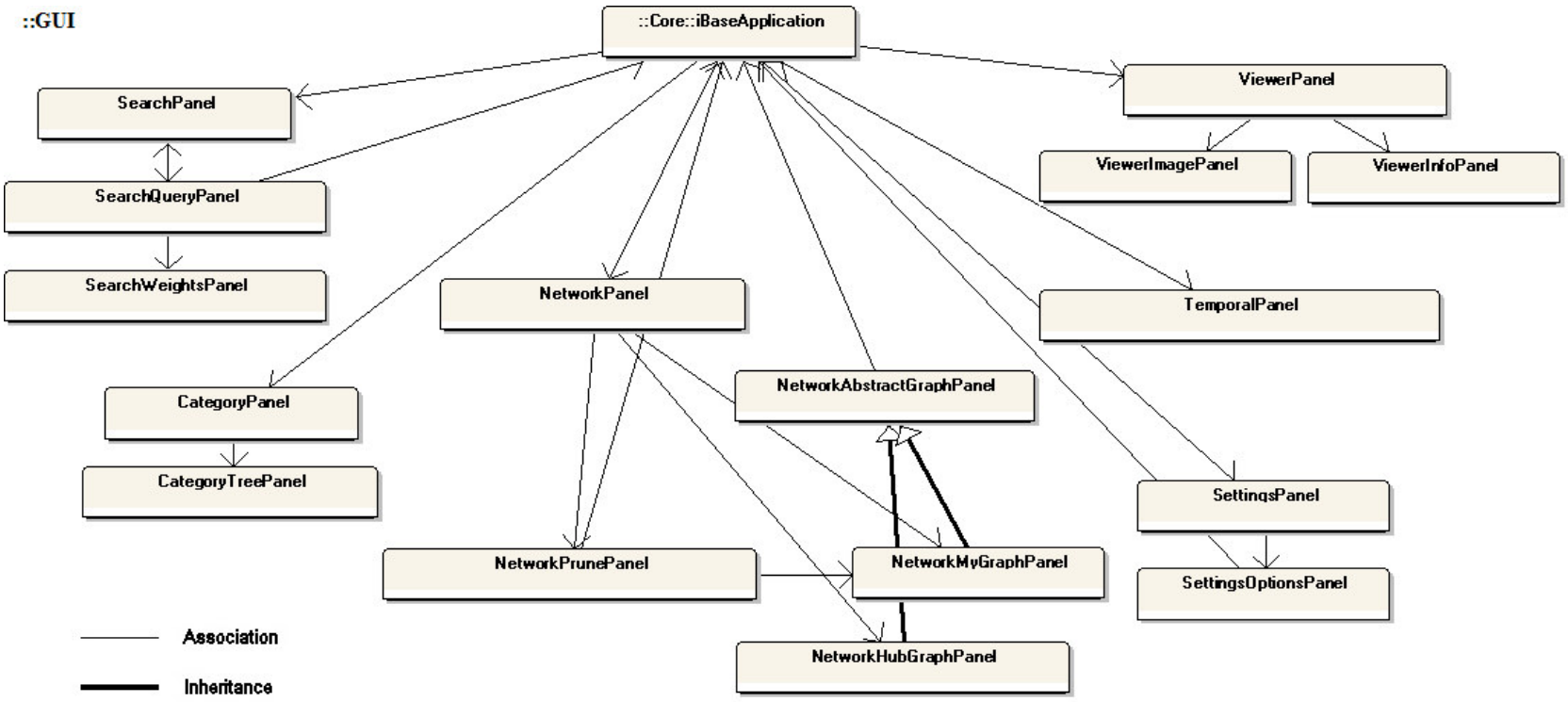
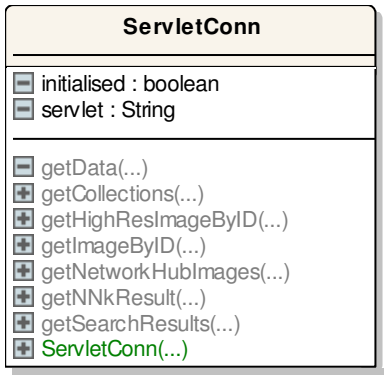
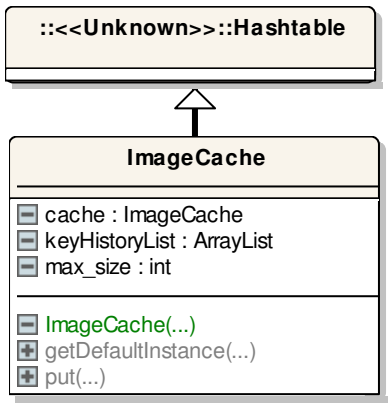


Figure 30. Applet class interaction



The applet communicates with the servlet through the `ServletConn` class. Any class in the applet can construct a new `ServletConn` and call the relevant method. This packages the request and parameters into a `Vector` and sends it to the servlet as a serialized object. The `ServletConn` class then unpackages the response from the servlet, and returns the approriate data type to the caller.



Each time the applet requests an image from the servlet by calling `getImageByID(collectionID, imageID)`, we first perform a look up in the image cache to see if that `Thumbnail` object has been retrieved already. If it has, we return the caches `Thumbnail`. The `ImageCache` is implemented as a `Hashtable` indexed with a 64-bit key of type `long` computed as:

$$2^{32} * \text{collectionID} + \text{imageID}$$

We impose a (user-configurable) limit on the cache size so that we control memory usage. If a `Thumbnail` is not in the cache, then it is fetched from the server and copied to the cache by the `ServletConn` class before being returned to the caller.

5.2.2 Image search and browsing of results

The `SearchQueryPanel` class is responsible for formulation of the search query and initiation of a search. When the search button is pressed or the `search()` method is called, a new `ServletConn` object is instantiated, and the `getSearchResults()` method called with the query parameters. Then a new `FetchResultsImagesThread` is created and passed the list of search results and a reference to the `ResultsPanel`. This thread takes the time-consuming task of retrieving the `Thumbnail` of each search result away from the event-dispatching thread (the thread that executes code when the user triggers an action through the interface). This ensures that the interface remains responsive while we fetch images from the server. We are also able to

update the interface after each image is retrieved, so that the user does not have to wait for the whole set of results to be retrieved. The user can stop or refresh the thread using the toolbar buttons, which invokes a call to stopImageThread() or refreshImageThread(), which gets cascaded down to the thread.

The `FetchResultsImageThread` locally selects the current 'image of interest' if it appears in the results. If it does not appear, then the first image in the results is globally selected (`setSelectedImage()` cascaded to all other browsing panels).

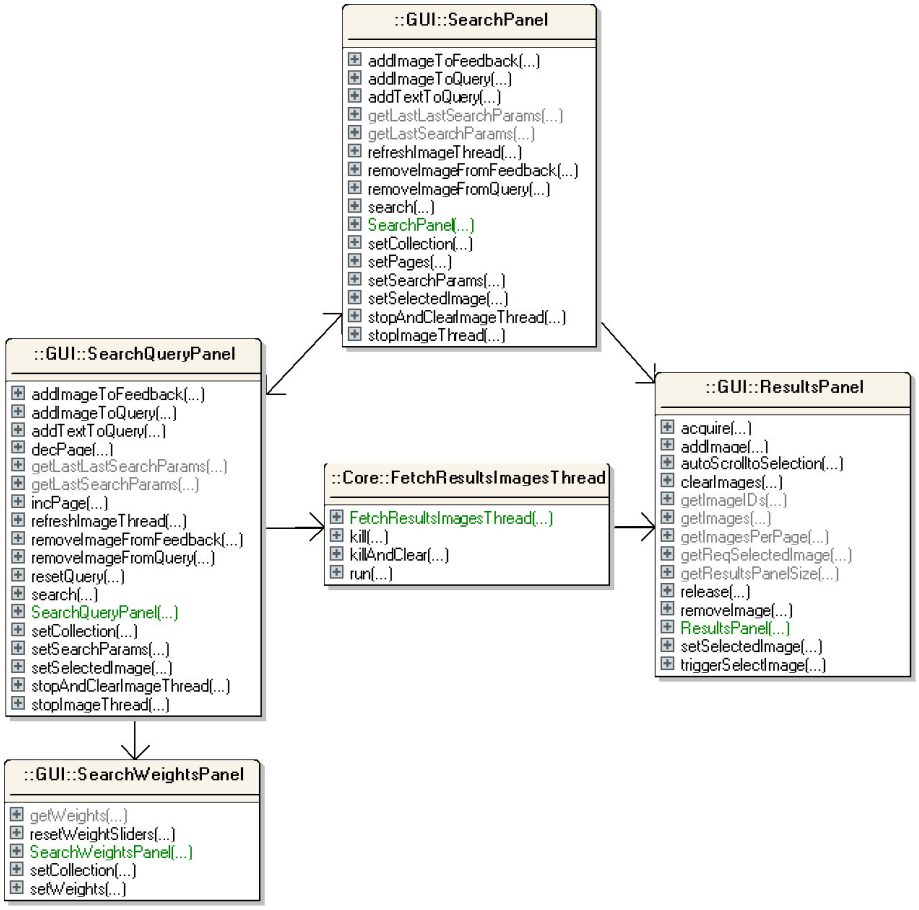


Figure 31. Interaction of classes involved in searching

5.2.3 Hierarchical browsing

The `CategoryTreePanel` class is responsible for constructing the tree interface component from the category name information of the currently selected `ImageCollection`. When the user selects a category from the tree, or a call to `setCategory()` has been cascaded from `iBaseApplication`, a new `FetchCatImagesThread` is created and passed the category path, a flag to say whether or not the change is user-invoked or system-invoked, and a reference to the `ResultsPanel`. Again, we instantiate a new thread to take the time-consuming task of retrieving `Thumbnail`s away from the event-dispatching thread to ensure responsiveness of the interface. The thread updates the `ResultsPanel` as each image is retrieved in the same way as the search results. If the new category selection is system-invoked (a result of a user selecting a new image in another panel) then this image is locally selected when it is retrieved. If the new category selection is user-invoked (a result of the user choosing the category from the tree interface) then the first image in the category is globally selected (triggers a change in all other panels).

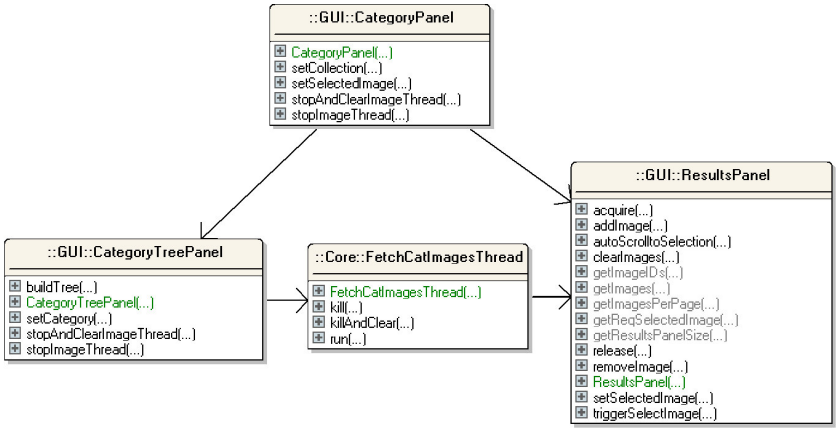


Figure 32. Interaction of classes involved in hierarchical browsing

5.2.4 Lateral browsing (NN^k networks)

The `NetworkPanel` constructs the three different panels involved in lateral browsing: the `NetworkPrunePanel`, `NetworkMyGraphPanel`, and `NetworkHubGraphPanel`. There is much common functionality between the panel displaying the hub network and the panel displaying the user network, which we have abstracted to the class `NetworkAbstractGraphPanel`. When the user selects an image collection, a call to `setCollection()` is cascaded from `iBaseApplication`. This invokes the instantiation of `FetchNNKHubImagesThread` which retrieves the hub images from the server.

The `NetworkPrunePanel` is responsible for drawing the nearest neighbours of the currently selected image in a spiral. When a call to `setSelectedImage()` is received, `FetchNNKNeighbourImagesThread` is instantiated. This fetches all neighbour images of the currently selected image and updates the interface as they are retrieved by calling `addImage()` on the `NetworkPrunePanel`.

After the user has selected which images are relevant and presses the 'prune' button, the `NetworkPrunePanel` calls `addImage()` on the `NetworkMyGraphPanel` to add the pruned neighbours to the user's graph, where they can be manipulated. By selecting an image in the graph, the prune panel is updated to show the neighbours of that image via the cascading `setSelectedImage()` mechanism.

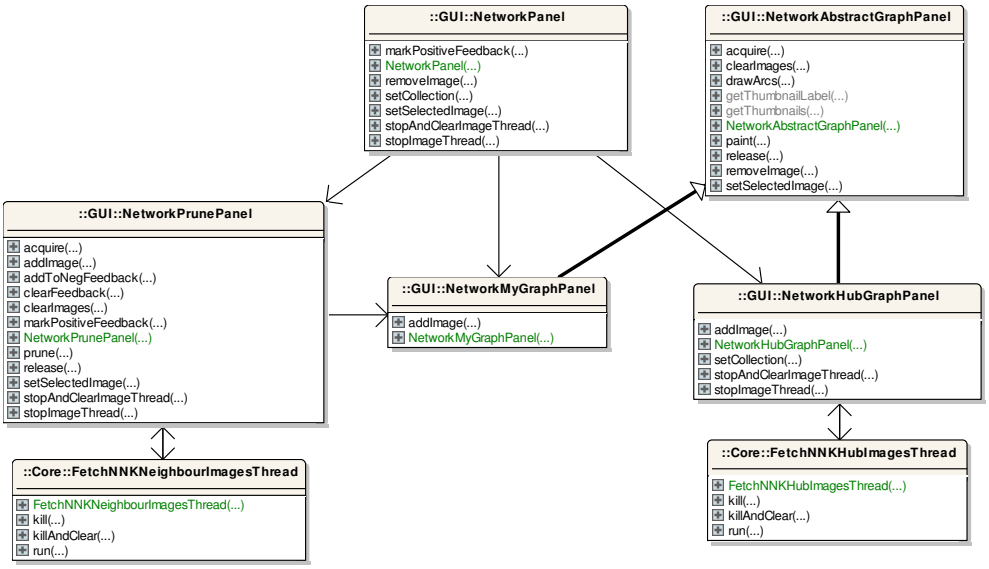
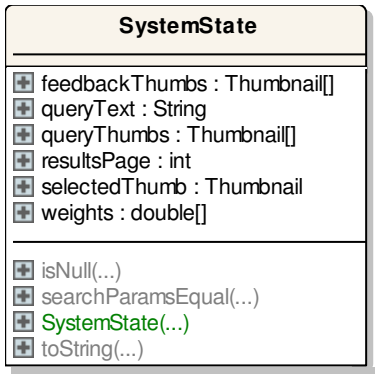


Figure 33. Interaction of classes involved in lateral browsing

5.2.5 Historical browsing

Every time the current 'image of interest' is changed, a call is made to `setSelectedImage()` in `iBaseApplication`, so that the change is cascaded to all other browsing panels. Therefore this is the perfect place to implement historical browsing. Each time the selected image is changed, we can log the previous image in our history of image navigation. Recall our notion of historical browsing discussed in 4.6. As well as the last selected image, we must also store the search parameters at the point when that image was selected, so that we can reconstruct the search results.



To achieve this, we defined a data type `SystemState` which stores all necessary information about the state of a system. This method makes it easy to extend our notion of the system state in the future. We use two `ArrayLists`, `backwardHistory` and `forwardHistory`, to keep track of `SystemState` objects. Below we have given the pseudo-code operations required to maintain a history of the system state:

On change of the selected image:

- Clear forwardHistory
- Instantiate a new SystemState object with current search parameters and current selected image
- Add SystemState object to backwardHistory
- Change to new image

On press of 'back' button:

- Instantiate a new SystemState object with current search parameters and current selected image
- Add SystemState object to forwardHistory
- Remove last SystemState object from backwardHistory, restore search parameters and select image

On press of 'forward' button:

- Instantiate a new SystemState object with current search parameters and current selected image
- Add SystemState object to backwardHistory
- Remove last SystemState object from forwardHistory, restore search parameters and select image

5.2.6 Temporal browsing

The `TemporalPanel` class is responsible for the temporal sequence of images. When `setSelectedImage()` is called, a `FetchTemporalNeighbourImagesThread` is instantiated. This retrieves the left and right temporal neighbours of the selected image using an instance of `ServletConn`. It continues to retrieve the left temporal neighbour of the leftmost image in the sequence and the right temporal neighbour of the rightmost image in the sequence, until the (user-configured) required number is reached. As usual, the interface is updated with the thumbnail images as they are retrieved. By clicking on an image in the sequence, the user selects it as the 'image of interest' and the panel updates to show the temporal neighbours of that image. The other browsing panels also update accordingly.

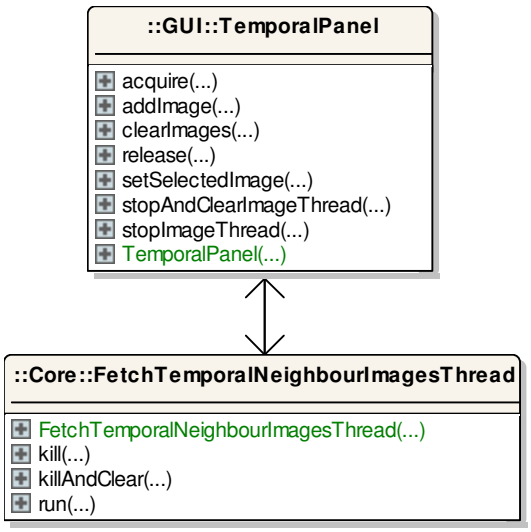


Figure 34. Interaction of classes involved in temporal browsing

5.2.7 Image viewer

The `ViewerPanel` class constructs the two panels of the image viewer interface: `ViewerInfoPanel` and `ViewerImagePanel`. When `setSelectedImage()` is called, a `FetchHighResImageThread` is instantiated. This fetches the high-resolution image from the server by calling `getHighResImageByID()` instead of the usual `getImageByID()`. When the thread has received the `Thumbnail` object, it calls `addImage()` on the `ViewerImagePanel` to draw the high-resolution image on the interface, and `setDetails()` on the `ViewerInfoPanel` to display the image's details.

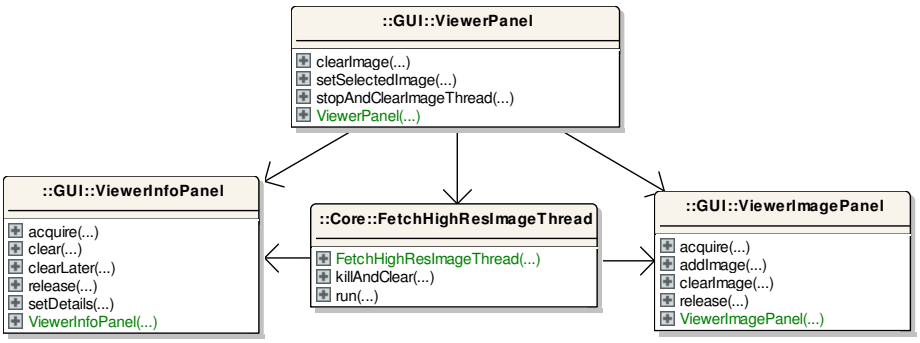


Figure 35. Interaction of classes involved in the image viewer

5.2.8 Client settings

The `SettingsPanel` class constructs and adds the `SettingsOptionsPanel` and `Log` to the interface. Settings are stored in a static `Hashtable`. Any class can set or retrieve settings by calling the static functions `iBaseApplication.setSetting(name, value)` and `iBaseApplication.getSetting(name)`. The SettingsOptionsPanel is responsible for the interface and functionality which allows the user to modify the program settings. On start-up, the image collection list is populated with all served image collections and the first image collection is selected. The feature list is populated with the implemented features of this collection and they are all selected by default. The program settings list is populated with the program setting default values. When the user selects an image collection and presses the 'Apply' button, it calls the `setCollection()` method on `iBaseApplication`. In a similar way to `setSelectedImage()`, `setCollection()` is cascaded to all browsing panels, which update

accordingly. The feature list is repopulated with the implemented features of the new collection. Selecting an image collection and selecting features updates the system settings "Selected Features" and "Current Collection" respectively. All other settings are directly modifiable by the user under the program settings interface. It is easy to make parts of the browser user-configurable by providing a new setting which can be modified on the `SettingsOptionsPanel`.

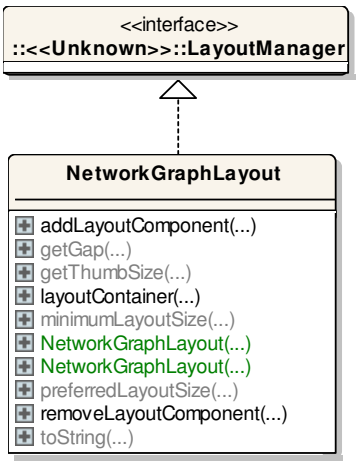
5.2.9 Other implementation issues

5.2.9.1 Swing layout managers

Due to the nature of the project, it is imperative that our design is not restricted by the limitations of our chosen programming language's GUI capabilities. One of the reasons for choosing Java was its flexibilty and power. Where the (already extensive) Swing library does not cater for our needs, we can extend Swing objects with our own implementations. One important area in which we have done this is in laying out interface components. Swing uses `LayoutManager`s to define how components should be laid out on a panel. In this way, we can ensure components scale gracefully when the user resizes the window or adjusts the allocation of space with a divider, making our browser 'screen-aware'.

We have implemented the following custom layout managers:

- `NetworkGraphLayout` (opposite) – all components in the panel scale in proportion to the panel size.
- `TemporalLayout` – Component 0 is added to the centre of the panel and is scaled so that the height of the component occupies all available space. Additional components are laid out alternatively to the left and right of the central component, decreasing in size in proportion to displacement from component 0, until there are no more components or the edges of the panel are reached.



- `PrunePanelLayout` – components are laid out in a spiral fashion from the centre. Component size is scaled so that the spiral occupies the available space.

- `ViewerLayout` – Component 0 is scaled to occupy the available space but retain aspect ratio. Additional components are ignored.

5.2.9.2 Thread safety

Thread safety is an important concern in any application where multiple threads access or update shared objects. We use threads to increase the apparent performance of the browser to the user by ensuring the event-dispatching thread is not blocked with time-consuming tasks. In our case, the output of our threads' time-consuming tasks is the retrieval of images which must be displayed on the interface. This poses a problem since Swing code is not thread-safe. That is, it does not inherently manage the problems of multiple threads executing the same code. In certain situations, multiple threads accessing shared methods and variables could conflict and cause errors or even deadlock.

To overcome this problem, Sun suggests that "all code that might affect or depend on the state of a component should be executed in the event-dispatching thread". The `SwingUtilities` class provides the static method `invokeAndWait(Runnable work)` for deferring work the event-dispatching thread.

Below we show the `addImage` method on `ViewerImagePanel`. The add operation needs to be executed by the `FetchHighResImageThread` when the image has been retrieved from the server. In order to make the method thread-safe, we must defer the code which adds the image to the interface to the event-dispatching thread by calling `invokeAndWait()`.

```
public void addImage(ThumbnailLabel tl) {
   if (!SwingUtilities.isEventDispatchThread()) {
        final ThumbnailLabel thumbLabel = tl;
        Runnable addImage = new Runnable() {
                public void run() {add(thumbLabel); updateUI();}
        };
        SwingUtilities.invokeAndWait(addImage);
   } else {
        add(tl); updateUI();
   }
}
```

By applying this technique to all methods which affect or depend upon interface components, we guard against thread conflicts while executing Swing code.

However, conflicting threads may still cause anomalies. Consider the situation where we wish to kill a thread that is adding images to a panel, clear the panel, and start a different thread to add images to that panel. Because threads are independently executing entities, we do not know in which order these actions will take place. The previous thread may not stop before the next has started, and so we would see an incorrect mixture of images in the panel. To overcome this problem, we place locks on shared interface components so that only one thread can act on a particular component at any time. Before a thread can act on an interface component, it must `acquire()` its lock, and `release()` it when it has finished updating to allow waiting threads to execute. Furthermore, we queue waiting threads in order, to ensure threads act upon the component in the correct order. The Java code for the lock mechanism is shown below.

```
public synchronized void acquire() {
   try {
        lockReqQueue.enQueue(Thread.currentThread());
        while (lockReqQueue.front()!=Thread.currentThread()) wait();
        notifyAll();
   } catch (InterruptedException ie) { }
}

public synchronized void release() {
   try {
        while (lockReqQueue.front()!=Thread.currentThread()) wait();
        lockReqQueue.deQueue();
        notifyAll();
   } catch (InterruptedException ie) { }
}
```

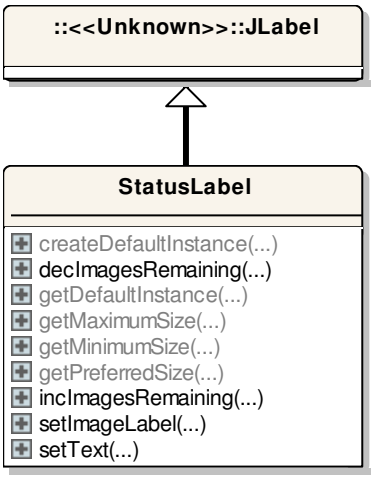
5.2.9.3 Pop-up menus

Each image displayed on the interface is a `ThumbnailLabel`. This inherits from `JLabel` and is essentially a Swing wrapper for the `Thumbnail` data type. Associated with each `ThumbnailLabel` is a `ThumbnailLabelMenu`. This inherits from `JPopupMenu` to provide the menu that appears when the user right-clicks on an image. When we instantiate a `ThumbnailLabel`, we must declare in which interface context the image resides. This enables the `ThumbnailLabel` to construct a `ThumbnailLabelMenu` that is context-aware, i.e. provide only those options which are relevant for the browsing interface in which the image is displayed.

5.2.9.4 Custom tooltips

In order to draw the pie charts, we create our own tooltip class, `ThumbnailToolTop`, which inherits from `JToolTip`. We then override the `paint()` method so that the pie chart is drawn with the features similarity contribution information. We must also override the `createToolTip()` method of `ThumbnailLabel`, to return a `ThumbnailToolTip` instead of a `JToolTip`.

5.2.9.5 Statusbar



To keep the user fully informed as to the total number of images that are still to be retrieved, we implemented a custom status bar by inheriting from `JLabel`. Of course this is a shared resource, so we must ensure access is thread-safe.

A thread can add to the number of images remaining by calling `incImagesRemaining(int number)` and similarly decrement from the number of images remaining by calling dec`ImagesRemaining(int number)`. Each time these methods are called, the total remaining is adjusted and the statusbar interface component is updated. If there are no images remaining, the statusbar displays the details of the currently selected image.

5.3 The servlet

The servlet component of the browser is run in an application server environment such as Apache Tomcat. The application server maps a URL on the local machine to the `doPost` method of the Java program (`HttpServlet`). When an HTTP request is received by the server on the appropriate URL, the `doPost` method is called, passing references to the `HttpServletRequest` and `HttpServletResponse`. The servlet takes the request, performs some processing and then writes its response to the `HttpServletResponse`. In our case, the `Server` class takes the request `Vector`, unpacks the arguments and calls the appropriate method on `ImageCollectionReader` or `ImageCollectionSearcher`. It then packages the response into a `Vector`, and writes this to the `HttpServletResponse OutputStream`. See 5.1.6 for more information about the client-server interface.

Figure 36 shows the interaction between classes in the server. `ImageCollectionReader` and `ImageCollectionSearcher` are defined as interfaces to allow the underlying implementation to be changed transparently. If we had an image collection with a different file structure on disk to that described in 5.1.4, we could replace the implementing classes without affecting the rest of the system.

When the servlet is started by the application server, the `Server` class reads the server parameters from the **server.properties** file and instantiates a new `IBaseImageCollectionReader` and `IBaseImageCollectionSearcher`. On instantiation, `IBaseImageCollectionReader` scans the collections directory and for each collection reads in:

- The list of image filenames from **image.lst**
- The collection category names from **image.lst**
- The first image ID of each category from **image.lst**
- The list of implemented features from **config.xml**
- The temporal neighbour for each image from **neighs.txt**

If the Server's `doPost()` method is called with a `getCollections` request, it calls `getCollections()` on `IBaseImageCollectionReader` class to return an array of the served `Collection`s constructed from the data read in above.

If a `getImageByID` or `getHighResImageByID` request is received, the `getImageByID` method of `IBaseImageCollectionReader` is called. This performs the following steps:

- Reads the image data from the **data** directory (or **images** directory if high-resolution request)
- Reads the associated image caption from the **text** directory
- Reads the NN^k neighbours from the **browser** directory

- Looks up the temporal neighbours in this collection's temporal neighbours list
- Constructs and returns a new `Thumbnail` object with the above data

If a `getSearchResults` request is received, the `search()` method of `IBaseImageCollectionSearcher` is called. This performs the following steps:

- If there is query text, it instantiates `TextSearchService` and calls the `search()` method. This uses the external text search engine Apache Lucene to construct an array of `SearchResult`s. We also compute an array containing the distance of each image from the query text, according to the relevance score that Lucene assigns each result. This is used to combine a text-based search with an image-based search.
- If there are no query images, then the text search results are returned. If there are query images, then `ImageSearchService` is instantiated and the `search()` method called. If there is also query text, then text distances are passed. The `search` method reads in the distance to the query image(s) for each active feature and for each image in the collection. The distances for each feature (including text) are combined into a total distance according to the feature weightings that have been passed from the client. The distances for each image are then sorted, and the requested interval (depending on the results page number and size of page) of results is returned to the client.

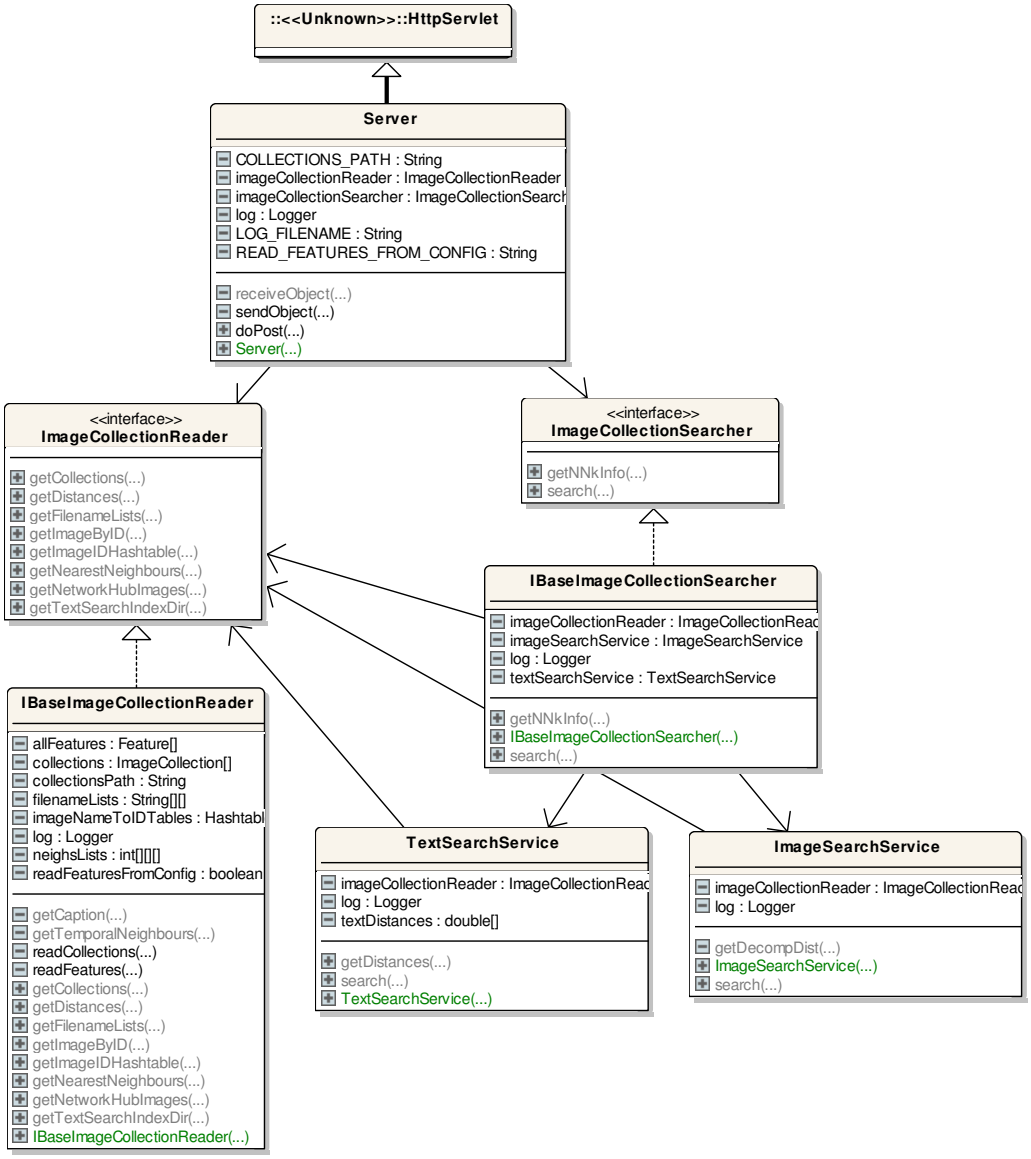


Figure 36. Servlet class interaction

6 Evaluation

There has been much research and discussion into formal evaluation methods for information retrieval systems. We shall first discuss the need to evaluate the quality of the search results in terms of document relevance. The first problem we encounter in establishing a formal evaluation metric is that of defining relevance. Relevance is subjective to the user of an information retrieval system and so is hard to formalise for evaluation purposes.

Ideally, an evaluation metric should be based upon an expert's judgement of relevance for each document. To limit the effects of differing opinions of relevance, TREC uses the following working definition of relevance: "If you were writing a report on the subject of the topic and would use the information contained in the document in the report, then the document is relevant." There is also a problem when a collection is so large that it is impractical for humans to examine every document for relevance. In these cases, and as adopted by TRECVID, a subset of the collection is presented for relevance judgements.

The most established and widely-used performance measures are those of *precision* and *recall*. They are defined as follows:

$$precision = \frac{No.\ relevant\ documents\ retrieved}{Total\ no.\ documents\ retrieved}$$

$$recall = \frac{No.\ relevant\ documents\ retrieved}{Total\ no.\ relevant\ documents\ in\ collection}$$

These are set-based measures that assess the quality of the set of documents retrieved and do not place any significance on the ordering of relevant documents within the results. This limits their usefulness when computed as single values. A more informative presentation of overall system performance can be provided with a precision-recall graph.

In order to evaluate a ranked list of results we can compute *average precision*. This corresponds to the area under an ideal (non-interpolated) recall-precision graph. It is calculated by taking the precision after every retrieved relevant document and then averaging these precisions over the total number of relevant documents in the collection. In this way, we favour highly ranked relevant documents. To test performance of a system over many different queries (known as 'topics'), we can take the average of the average precisions

over the number of topics. This is known as the mean average precision (MAP) and forms the basis of the TRECVID evaluation.

Several other measures which are based on precision and recall are also used in TREC, as summarized in [26]:

- P(10), P(30), P(NR) – the precision after the first 10, 30, NR documents are retrieved, where NR is the number of relevant documents for this topic.
- Recall at 0.5 precision – recall at the rank where precision drops below 0.5.
- R(1000) - recall after 1000 documents are retrieved.
- Rank first relevant – the rank of the highest-ranked relevant document.

Performance measures allow us to easily compare system results but do not provide us with any direct measure of usability. This is obviously of utmost importance in a user-oriented application such as a browser where user interaction is so important. Indeed it has been argued that in many cases usability is more important than performance. No user will want to use an application which is inflexible, unresponsive or non-deterministic, irrespective of performance. In evaluation we shall aim to address Nielsen's [27] five usability attributes: learnability, efficiency, memorability, errors and satisfaction.

Usability evaluation is extremely subjective and as such difficult to quantify. One method of gauging strengths and weaknesses in usability is to use a questionnaire. The effectiveness of questionnaires as an evaluation method lies in the careful formulation of its questions. We must be sure to address the evaluation questions asked, and avoid introducing any bias. We must also ensure that the questionnaire is quick and easy for a user to complete.

6.1 Performance analysis

6.1.1 Creating a ranked list output

Until now we have considered the outcome of the user's browsing and searching to be the set of images added to the 'My Selection' panel. These are results that the user has assessed as relevant and individually added to the panel. However, it is possible that the user might want to find the best 100 or 1000 images that match certain criteria. In these cases, it is not feasible for the user to assess each individually for relevance and add to the output accordingly.

For this purpose, and to assist performance analysis, we have added a panel to the application on which the user can create a ranked list of images as 'output' from the searching and browsing process, using the 'My Selection' panel and the search results as sources. Since all images added to 'My Selection' have been individually assessed by the user, it seems reasonable that these will be relevant and so can be added to the start of the output list. We then append search results until the required number of images is met, ensuring there are no duplicated images.

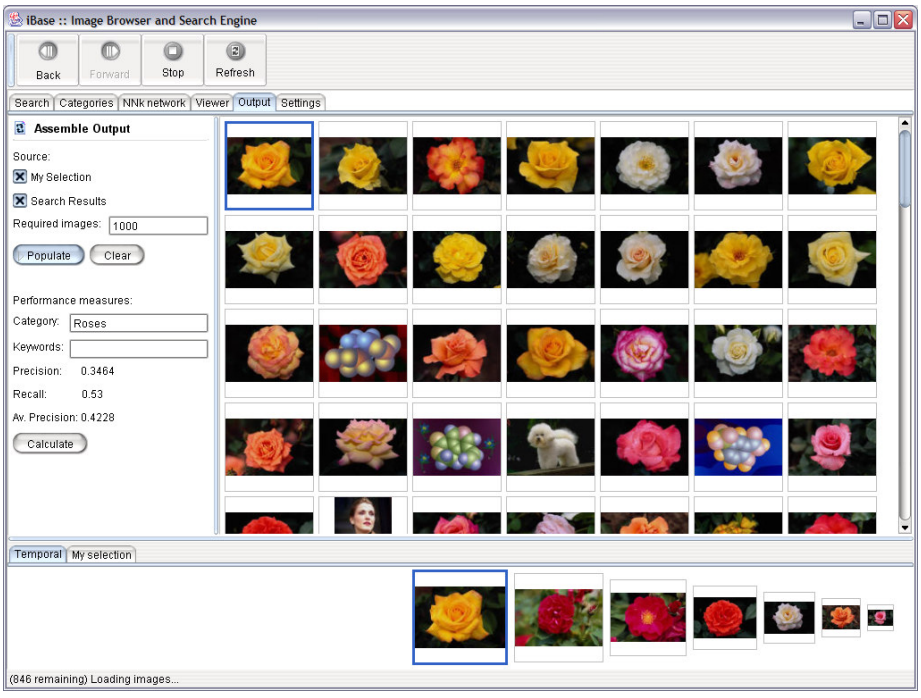


Figure 37. Assembling the ranked list output

We also provide the user with the ability to manually rearrange the order of the output list of images. This is performed using a slide-sorter style drag and drop mechanism. This is an established and intuitive method for rearranging a list of objects and has been used for years in presentation packages. This functionality allows the user to perform some final 'tweaking' of results where obvious improvements could be made.

6.1.2 Experiments

We discussed above the common performance metrics used. We shall evaluate the retrieval power of the application in terms of recall-precision graphs and mean average precision, in the same fashion as TRECVID evaluate image retrieval systems. However, we do not have the resources to manually evaluate each image in the output list for relevance, so we shall adopt an automated approach suggested in [28]. This involves using an image collection in which each image has been assigned a category by a human. We can therefore take our notion of relevance as membership of a particular category. So if we search for penguins then we can determine if an output images is relevant by testing for membership of the 'penguins' category.

We shall use a subset of the Corel Gallery 380,000 photograph collection which contains a total of 6,162 images split into 63 categories. Topics were chosen by selecting 8 categories which showed both some intra-category visual similarity and some higher-level semantic meaning. This should appeal to the strengths of the content-based search engine and the browsing methods respectively. For each of the 8 categories we selected 3 query images at random. This gave us the following topics:

Topic	Category	Size	Query image IDs
T1	African Antelope	99	24, 56, 93
T2	Bears	99	413, 420, 457
T3	Bridges	90	803, 838, 871
T4	Castles of Europe	82	1190, 1242, 1263
T5	Coastal Landscapes	72	1471, 1489, 1512
T6	Contemporary Buildings	89	1593, 1617, 1630
T7	Penguins	99	4522, 4526, 4544
T8	Spectacular Waterfalls	94	5620, 5626, 5673

In order to investigate the interaction between the different searching and browsing techniques used, and their relative contributions to the quality of the results, we conducted the experiments under four system variants. This is an approach that was employed successfully for Imperial's entry to TRECVID 2003 [10].

I Search + NN^k browsing + Relevance feedback
II Search + NN^k browsing
III NN^k browsing only
IV Search + Relevance feedback
V Search only

Text search and temporal browsing were disabled for the experiments, since they are based upon human annotation and human categorization respectively. The search-only experiment is not subject to any user interaction and was computed for comparison purposes.

To test the performance under the four interactive system variants we used a latin square method in which we use four users and the 8 topics are divided into pairs:

	T1/T2	T3/T4	T5/T6	T7/T8
U1	I	II	III	IV
U2	IV	I	II	III
U3	III	IV	I	II
U4	II	III	IV	I

In this way, we minimise the effect of searcher and topic learning on the resulting performance measure. Each user had at least 30 minutes prior use of the system with a different image collection, to become acquainted with the techniques used. They were then given 10 minutes to assemble a ranked output list of 1000 images for each system variant. In each case, we computed the interpolated precision values for a set of standard recall levels (0 to 1 in increments of 0.1) standard as described in [29]. This facilitates the computation of average performance over a set of topics with different numbers of relevant documents. For each scenario, we also computed the (non-interpolated) average precision which was averaged over all topics to give the mean average precision of each system variant.

6.1.3 Results

Full experiment results are given in the Appendix. Figure 38 shows the precision-recall graphs for each system variant averaged over the 8 topics. Figure 39 shows the corresponding mean average precision values.

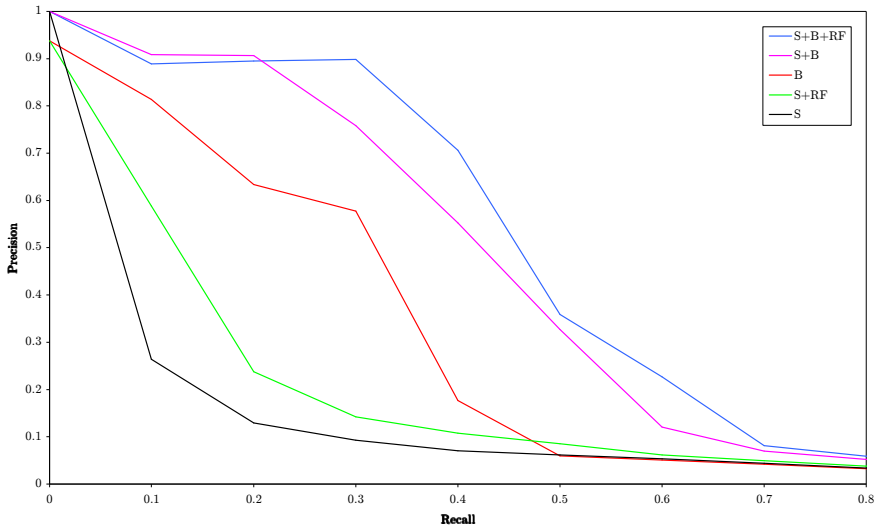


Figure 38. Precision-recall graph for each system variant (averaged over all topics)

System variant	Mean average precision
S+B+RF	0.4269
S+B	0.4072
B	0.2672
S+RF	0.1687
S	0.1112
Random list	0.0035

Figure 39. Mean average precision for each system variant

6.1.4 Discussion

It is clear from the results that by integrating the different techniques of searching, browsing and relevance feedback we gain a substantial improvement in the quality of the ranked list of images. Relevance feedback (including query modification) provided us with a 52% increase in mean average precision (MAP) over a standard search. However it proved less significant when browsing and searching were used together, giving a mere 4% increase in MAP.

Perhaps the most surprising of all the results is the performance gain through NN^k browsing. On average, browsing boosted a system variant's MAP by over 300% relative to the equivalent non-browsing system variant. We can explain the magnitude of this increase in several ways. It might be attributed to the relatively small size of the image collection and relatively large category sizes. Since there are 6,192 images and 63 categories, on average approximately 1 image for every 100 examined will be relevant. This means that the user can find a relevant image relatively quickly through browsing. Once a relevant image is found, the user can often go on to uncover many more relevant images by exploring the lateral neighbours. The size of the image collection might also mean that the NN^k network is more tightly connected than in a larger collection.

Our category choices might also play a significant role in determining the relative performance of searching and browsing. For topics in which category images are visually similar (eg. penguins), searching performed well. However where little visual similarity exists (eg. contemporary buildings) searching did not perform well. It is in these cases where we can see the benefits of the NN^k browsing structure. By exposing the semantic richness of an image to the user we can elicit a user's high-level notion of relevance.

By combining all three methods, we can gain an almost fourfold improvement in mean average precision over a standard feature-based search. However, we must understand the limitations of our evaluation technique. The Corel categorisations where not designed for use in such relevance tests, and it means we are locked into one fixed notion of relevance for each image. This causes inaccurate relevance assessment in the case where an image can fall into more than one category. Our evaluation was also limited by time and resources. Although our initial testing suggests there are large gains to be had from combining retrieval methods, the system should be subject to real-world queries involving many topics and whose results are assessed for relevance by humans such as in the TRECVID evaluation. This would allow us to gain more compelling evidence for the benefits of the system.

6.2 Usability study

As part of the system evaluation, we conducted a questionnaire-based usability study. We based our method on the general usability questionnaire suggested in [30]. This uses a system similar to the "Likert scale" where the subject indicates to what extent they agree or disagree with a statement about the system. The statements are constructed so that half are positive in tone (odd numbers) and half are negative in tone (even numbers). They have also been designed to address Nielsen's [27] five usability attributes. A sample questionnaire is shown in the Appendix.

Each user responds to the questions with a rating as follows:

Rating	Strongly agree	Agree	Undecided	Disagree	Strongly disagree
Positive statement	5	4	3	2	1
Negative statement	1	2	3	4	5

In this way, a higher score indicates greater usability.

Before completing the questionnaire, each user was given a 15 minute demonstration of the system and then was allowed (at least) 15 minutes to use the system without assistance. A relatively small sample of 6 users participated in the study due to time and resource constraints. Ideally, we would maximise the sample size within practical limits. The users were all of the age range 21-23 and their knowledge of the information retrieval domain extended no further than everyday use of operating systems and Internet search engines.

Figure 40 shows the score assignment to the questionnaire responses. For each question we have computed the sum of the scores, and then ordered from highest score to lowest. A high score indicates greater usability. We can see that the speed and responsiveness of the system was highlighted as a strong point. This might be attributed to our careful use of threads to ensure time-consuming tasks can run the background so that the interface remains responsive and the user can continue with other tasks. Other strengths include lack of frustration and a feeling of user control. This is due to the rich set of user actions available and the standard browser "Back", "Forward", "Stop" and "Refresh" buttons which is a familiar paradigm that creates a sense of complete control over navigation.

	Statement	Score (max 30)
18	The system was slow and unresponsive	27
14	I often felt frustrated when using the system	27
15	The system's functions were well integrated	26
13	I was always in control of what the system was doing	25
2	The system was difficult and cumbersome to use	25
7	The system was easy to use	24
16	Mistakes were difficult to rectify	24
9	System functions were easy to remember	23
5	The system performed in the way I expected it to	23
17	I never felt limited by settings that I could not modify	23
1	I would choose to use this system again	22
3	I felt confident when using the system	22
12	It was difficult to remember icon/button functions	22
19	I enjoyed using the system	22
10	The system was too complicated	21
11	Most people could use this system efficiently with no problems	19
20	System feedback was not helpful	18
4	I could not use this system without some form of technical support	17
6	I was often unsure as to what action I should take next	17
8	I would need much practice before I could use this system competently	14
	Total (max 600)	**441**

Figure 40. Usability questionnaire results

It was pleasing to see that users rated the integration of the system highly. This has been one of the main goals of the project and appears to have been achieved successfully.

General questions regarding ease of use and user satisfaction were ranked in the middle of the table with reasonably good scores. Questions which achieved the lowest scores appear to be those concerning the prior knowledge required by users. Because the user-base had limited knowledge of information retrieval systems, and only a relatively short demonstration of the system was carried out, there was a feeling that users would need more help and practice to use the browser effectively. Some users felt they could not use some functions as they did not know enough about the underlying searching and browsing mechanisms. This is a problem that faces any implementation of new and novel navigation techniques. Although we have tried to make the browser as easy to use as possible, the user does need some knowledge of the browsing and searching processes. Currently there is limited online help, which explains why users might feel unsure on how to proceed in their navigation.

There is a clear dilemma here. We must strike a balance in the mechanisms which we expose to the user and which we make transparent. There is a trade-off between simplicity of use and the user's level of control or navigational power. For example, we could have decided to make the concept of feature weights transparent to the user. However, this would limit the user's control over the searching process. There are some ways in which we can limit the problem of prior knowledge requirements and increase learnability while retaining navigational power. These include increasing the level of online help available and using more user-friendly terminology and analogies. For example, we could rename our feature descriptors to "Colour", "Texture" and so on.

The overall score of 441/600 seems positive. However, it is difficult to draw many useful conclusions from this total usability rating. This value might be useful in the comparison of different interfaces which have been assessed under the same criteria.

7 Conclusions

7.1 Achievements

We have devised, implemented and evaluated a web-based image browser and search engine which encompasses the following techniques in information retrieval:

- **Text-based search** – search using manual annotations or speech recognition transcripts to find relevant images in a collection
- **Content-based search with relevance feedback** – search using a combination of feature descriptors to match query images to images in a collection. We use relevance feedback to optimise the feature weightings
- **Hierarchical browsing** – browsing of image categories which have either been generated manually or arbitrarily
- **Lateral browsing (NN^k networks)** – browsing across the feature weight space by our notion of an NN^k network
- **Temporal browsing** – browsing in the temporal dimension, derived from video sequence information or timestamps
- **Historical browsing** – browsing through the history of the navigation path

Through careful analysis of the browsing and search processes, we have maximised user interaction to confront the problems of semantic gap and polysemy associated with content-based image retrieval. We have maintained sound HCI principles throughout with regard to consistency, responsiveness, progress information and other interface design issues. We have developed the software using object-oriented methodologies and common design patterns to ensure it is easily maintainable and extensible in the future.

Initial evaluation of the system is promising. It appears that large gains in performance can be achieved by integrating different paradigms in image retrieval to enable the user to take advantage of the merits of each. Further performance analysis, for example at the TRECVID evaluation, would add weight to this claim. Usability analysis is also encouraging. More extensive evaluation could be carried out to identify usability in comparison with other retrieval systems so that we can identify possible interface improvements.

7.2 Further work

Besides further evaluation, we have identified several areas which could be investigated in future work:

- **Leaning –** an interesting area which we have not mentioned is the notion that the system can learn from the user interaction that takes place. For example, if some neighbours are consistently pruned off in the NN^k network then it is probable that they bear no real semantic relationship with the query image and they could be permanently removed from the network structure.
- **Multiple selection –** in many cases it would be advantageous to be able to multiply-select images so that the same action can be applied to many images at the same time. This would be particularly useful for adding relevant images to 'My Selection' without having to add each image individually.
- **Bandwidth-awareness –** although the current implementation employs caching and pre-fetching, the browser has no concept of the available bandwidth. For a modem user it may be useful to be able to disable pre-fetching, so that only images that are on the currently viewer panel are downloaded from the server. A more sophisticated implementation might detect available bandwidth and retrieve thumbnails at an appropriate size.
- **Add an external image to a collection –** it is useful for a user to be able to add an image to the collection on-the-fly. This poses some implementation challenges since currently image collections are indexed in advance to compute the necessary feature and browsing structure data.
- **Edit or add image annotations –** a useful feature might be the ability to edit or add annotations to an image collection.
- **EXIF tags –** further image information could be extracted from images by means of the Exchangeable Image File Format (EXIF) tags. Digital cameras often store extra information with an image which could be retrieved and displayed in the viewer.

References

[1] M. Pickering, D. C. Heesch, R. O'Callaghan, S. Rüger and D Bull. Video Retrieval using Global Features in Keyframes. *Proceedings of TREC* 2002, *NIST (Gaithersburg, MD, Nov* 2002*)*, NIST Special Publication 500-251, pp 318-324, 2003

[2] C. Naster. Content-Based Image Retrieval: A State of the Art. LTU technologies.

[3] S. Siggelkow. Feature Histograms for Content-Based Image Retrieval. MSc Thesis, Luneburg 2002

[4] D. C. Heesch and S. Rüger. Performance boosting with three mouse clicks – relevance feedback for CBIR. In *Proceedings of the European Conference on IR Research 2003*. LNCS, Springer, 2003.

[5] D. Daneels, D. Campenhout, W. Niblack, W. Equitz, R. Barber, E. Bellon, and F Fierens. Interactive outlining: an improved approach using active contours. *In Proceedings of SPIE Storage and Retrieval for Image and Video Databases*, 1993

[6] W. Y. Ma and B. S. Manjunath. Texture features and learning similarity. *In Proceedings of IEEE Conf. Computer Vision and Pattern Recognition*, pages 425-430, 1996

[7] T. P. Minka and R. W. Picard. Interactive learning using a society of models. *In Proceedings of IEEE Conf. Computer Vision and Pattern Recognition*, pages 447-452, 1996

[8] Guidelines for the TRECVID 2003 Evaluation.
http://www-nlpir.nist.gov/projects/tv2003/tv2003.html

[9] C. E. Jacobs, A. Finkelstein, and D. H. Salesin. Fast multiresolution image querying. In *ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 277–286, Los Angeles, CA, August 1995.

[10] D. C. Heesch, M. J. Pickering, S. Rüger, and Alexei Yavlinsky. Video Retrieval using Search and Browsing with Key Frames. *Proceedings of TREC Video Retrieval Evaluation(TRECVID, Gaithersburg, MD, Nov* 2003*)*.

[11] K. Tieu and P. Voila. Boosting image retrieval. *In* 5^{th} *International Conference on Spoken Language Processing*, Dec. 2000

[12] I. H. Witten, A. Moffat, T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann 1999

[13] D. C. Heesch and S. Rüger. Combining Features for Content-Based Sketch Retrieval - A Comparative Evaluation of Retrieval Performance. *Proceedings of the 24th European Colloquium on Information Retrieval Research (ECIR, Glasgow, 25-27 Mar 2002)*, LNCS 2291, pp 41-52, Springer-Verlag, 2002

[14] G. Z. Yang and D. F. Gillies. Shape Recognition Lecture Notes, Computer Vision Course, Dept of Computing, Imperial College

[15] BINS static image browser. http://bins.sautret.org

[16] WildlifeLands image archive.
http://www.gironet.nl/home/cdon/home.htm

[17] Microsoft Clip Organiser (a component of Microsoft Office).
http://office.microsoft.com

[18] The IDIAP Video Browser.
http://www.idiap.ch/~guillemo/videobrowser.html

[19] D. Heesch and S. Rüger. NNk networks for content-based image retrieval. *Accepted for publication in the Proceedings of the 26th European Conference on Information Retrieval (ECIR, Sunderland, UK, Apr 2004).*

[20] Google Image Search. http://images.google.co.uk

[21] M. Rautianen, K. Noponen, M. Hosio, T. Koskela, J. Liu, T. Ojala and T. Seppänen, J. Penttilä, P. Piertarila, S-M. Mäkelä, J. Peltola. TRECVID 2003 Experiments at Media Team Oulu and VTT

[22] M. Worring, G.P Nguyen, L. Hollink, J. van Gemert, D.C Koelma. Interactive Search Using Indexing, Filtering, Browsing and Ranking

[23] P. Browne, G. Gaughan, C. Gurrin, G.J.F. Jones, H. Lee,
S. Marlow, K. McDonald, N. Murphy, N.E. O'Connor, A.F. Smeaton, J. Ye. CDVP & TRECVID-2003 Interactive Search Task Experiments

[24] D. Albertson, J. Mostafa, J. Fieber. Video Searching and Browsing Using ViewFinder : Interactive Search Experiment for TRECVID 2003

[25] TREC Video Retrieval Evaluation, NIST.
http://www-nlpir.nist.gov/projects/trecvid/

[26] H. Müller, W. Müller, D McG. Squire, T. Pun. Performance Evaluation in Content-Based Image Retrieval: Overview and Proposals.
http://vision.unige.ch/publications/postscript/99/VGTR99.05_HMuellerWMuellerSquirePun.pdf

[27] J. Nielsen. Usability engineering. Morgan Kaufmann, 1994

[28] M Pickering and S Rüger. Evaluation of key-frame based retrieval techniques for video. *Computer Vision and Image Understanding*, 92.1, 217-235, 2003

[29] TRECVID Common Evaluation Measures
http://trec.nist.gov/pubs/trec10/appendices/measures.pdf

[30] T.-S. Lai. CHROMA: A Photographic Image Retrieval System.
http://osiris.sunderland.ac.uk/~cs0sla/thesis/