Support Tools for Visual Information Management*

Gökhan Kutlu Bruce A. Draper J. Eliot B. Moss
Edward M. Riseman
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

Visual applications need to represent, manipulate, store, and retrieve both raw and processed visual data. Existing relational and object-oriented database systems fail to offer satisfactory visual data management support because they lack the kinds of representations, storage structures, indices, access methods, and query mechanisms needed for visual data. We argue that ***extensible visual object store****s offer feasible and effective means to address the data management needs of visual applications.* ***ISR4*** *is such a visual object store under development at the University of Massachusetts for the management of persistent visual information. ISR4 is designed to offer extensive storage and retrieval support for complex and large visual data, customizable buffering and clustering, and spatial and temporal indexing, along with a variety of multi-dimensional access methods and query languages.*

Index Terms: *visual information management, persistent object store, extensible visual object store*

*This work was supported in part by the Advanced Research Projects Agency (via U.S. Army TEC) under contract number DACA76-92-C-0041, (via TACOM) under contract number DAAE07-91-C-R035, and by the National Science Foundation under grant number CDA-8922572.

1 Introduction

A *visual application* is an application that manipulates visual data as part of its processing. With advances in image analysis, visualization, and video technologies, increasingly large amounts of digital visual data are being generated by visual applications in a tremendously diverse range of domains, such as geographic, astronomical, and environmental information management, engineering and scientific visualization, military intelligence, computer aided design and manufacturing (CAD/CAM), and medical imaging.

Visual data consumed in applications consist not only of raw sensory data such as images, but also processed data, such as the knowledge structures used in visual interpretation systems, and associated model data (such as CAD/CAM models). As the scale of visual applications grows, the need to efficiently process, store, and access the raw and processed data becomes more acute.

Typically, a large amount of the data generated in a visual application is needed for temporary use only. A line extraction algorithm, for example, may produce hundreds (or thousands) of line segments from an image. Although not permanent, such data may be accessed repeatedly by other modules (e.g., line grouping or model matching algorithms), so the efficiency of in-memory data representation and retrieval

is critical.

On the other hand, *a priori* knowledge such as maps and models make up a permanent data base of information that visual algorithms access repeatedly and alter only occasionally. Similarly, generated long-term data, such as the set of extracted line segments that make up a site model, have to be stored for future access. Although less voluminous per image than the temporary data mentioned above, this data is persistent and grows to larger total amounts over time. It therefore must be managed by efficient storage and access mechanisms which are geared to the nature (e.g., spatial, temporal, 3D) of the data.

We addressed the management of temporary data in an earlier visual data management and process integration tool, called ISR3 [3]. In this paper, we first discuss the issues related to the management of persistent data in visual applications, and the shortcomings of current relational and object-oriented systems in dealing with these issues. We then argue that extensible visual object stores offer a feasible and efficient means to address the data management needs of visual applications, and present ISR4, a visual object store under development at the University of Massachusetts.

2 Current Technology

As described below, the efficient storage and retrieval of large volumes of permanent visual data, such as aerial images, site models, and MRI scans, imposes requirements that are vastly different from those found in conventional data processing. As a result, existing relational and object-oriented database systems fail to offer the kinds of storage structures, indexing and access methods, and query mechanisms needed for visual data.

2.1 Efficient Storage Structures

Large, Multi-dimensional Objects. One issue is how to manage the storage and retrieval of large, multi-dimensional objects such as images. Space- and time-efficient storage and access of large visual objects is critical in projects such as The National Digital Library program at the Library of Congress, which involves providing access to a major subset of approximately 105 million items, among which are large numbers of digitized pictures.

Most applications store images in files, and leave the management of memory (page swaps, etc.) to the operating system. This approach can result in a large number of page swaps, especially when the physical clustering of the image on disk does not match the access pattern of the application [18]. Traditional database systems do not provide appropriate data types or built-in support for images or similar 2D objects (e.g. maps). Adaptive clustering techniques used for clustering multi-dimensional data according to patterns of access are not mature, and the ones suggested depend on complex access pattern statistics [4, 18].

Associative clustering. As discussed above, many visual applications need to store not only raw images, but also symbolic data extracted from (or associated with) images. In content-based image retrieval, for example, commonly stored data include color histograms, invariants of shape moments, and texture features. Moreover, symbolic data often need to be associated with the image region they came from so that they can be retrieved with the sub-image. In the RADIUS [16] program, for example, site models reconstructed from sets of aerial images need to be grouped, stored, and retrieved according to their functional areas.

Most database systems provide little control over clustering of information in external storage so that a sub-image and the related analysis results can be stored on the same disk page, and retrieved together efficiently.

2.2 Multi-dimensional and Temporal Indexing

Visual data that are spatial in nature, such as geometric image structures, often need to be accessed according to their spatial properties in an image and/or 3D world positions. Therefore, spatial indices need to be maintained for efficient access to such data. Also needed, especially in military intelligence and medical imaging applications, are temporal indices defined over a time-sequence of image data. A typical medical query example is to find the first sign of a tumor in a history of MRI data.

Unfortunately, there is a lack of effective support for multi-dimensional and temporal indexing techniques in existing database systems. Moreover, simply adding one or two popular indexing methods, such as n-dimensional R-trees, is only a limited solution awaiting situations where a completely different index is needed. Instead, the ability to incorporate one's own indexing mechanism into the data management system is called for.

2.3 Query Mechanisms and Optimization

Spatial, temporal, and geometric representations. Current relational and object-oriented query languages do not express the necessary spatial, temporal, and geometric concepts and operators effectively. For example, one must usually build specific concrete representations of n-dimensional points, lines, curves, regions, etc., and most systems provide no appropriate treatment of geometric anomalies that arise from, for example, numerical roundoff errors. Although one can represent concepts such as points and lines, attempting to express notions such as "distance" and "collinearity" leads to very inefficient query processing in traditional database systems. Moreover, explicit coding of such data types sacrifices representational independence. Likewise, one typically does not have the most efficient algorithms available, e.g., from computational geometry.

Approximate, ranked retrieval. A deeper problem with existing query languages is that they are boolean. A fact or record either definitely lies in the query result set or it does not. Many queries in visual applications are more likely to be concerned with approximate matches and/or ranked retrieval, where the goal is to find the best answers to a query and to rank them according to their degree of quality. A simple example would be finding objects "near" another object: we might return a list of objects ranked according to their distance from the query object, up to some maximum distance and/or maximum number of objects.

Query Optimization. In addition to query languages being limited in concepts and operators, existing query optimizers are not prepared to take into account geometric algorithms, spatial/temporal indexing, and ranked retrieval. This may become a critical issue when scaling to large systems, since query optimization frequently has orders of magnitude impact on performance.

3 Extensible Visual Object Stores

A *visual object store* is an object store and its associated tools and facilities provided to support the representation, manipulation, storage and retrieval of visual data. An extensible visual object store will have a number of unique features, which help overcome the problems discussed in Section 2:

- It will provide a powerful core of functionalities to answer the basic data management needs of an application, including efficient built-in data types, basic storage and retrieval ability, and efficient storage structures, access methods, and query mechanisms for complex visual objects.
- Application programmers will be provided the flexibility to add new features as needed, at *all levels* of the system.

- Multiple policies and implementations will be available for the database implementor to choose from.
- Buffer management and data clustering policies will be accessible for customization and fine tuning.
- Applications will be lighter-weight since the features of the visual object store will be well integrated, and only those features needed will be part of the application; unnecessary features will be turned off.

Visual Information Management Systems. A similar research effort has focused on the development of Visual Information Management Systems (VIMS) [10]. However, there is so much variety in the application domains and the types of visual data they employ (e.g. continuous vs. discrete, temporal vs. spatial) that there is a need for a spectrum of VIMSs, rather than a single, all-encompassing VIMS. On the other hand, VIMS applications share a basic set of common needs; they all need to represent, manipulate, and effectively store and retrieve visual data. Addressing these mutual needs independently for each application would result in a duplication of effort. An extensible visual object store offers exactly what is needed for building application-specific VIMSs: support for their shared basic data management needs.

4 ISR4

ISR4 is an extensible visual object store under development at the University of Massachusetts. As shown in Figure 1, ISR4 is the integration of an earlier visual data management and process integration tool called ISR3[1] [3], with Mneme [14], a persistent object store (also developed at the University of Massachusetts).

[1] ISR (Intermediate Symbolic Representation; [Brolio, 1989]) is the name of a series of symbolic databases for visual information developed at the University of Massachusetts; ISR4 is the most recent version.

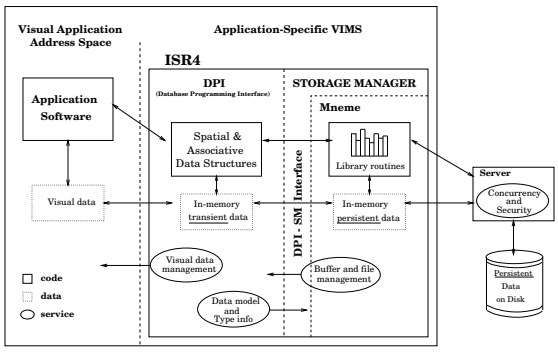

Figure 1: ISR4 is the integration of ISR3 and Mneme.

4.1 Overview of ISR4

Spatial and geometric representations. Embedded in its host language C++, ISR4 allows arbitrarily complex objects to be defined and processed, and provides an initial set of standard visual representations, including single-band and multi-spectral images, 2D points, lines, edges, and regions, and 3D points, lines, surfaces, and volumes. Moreover, as the (object-oriented) data model is uniform throughout ISR4, complex representations such as spatially-indexed sets of line segments or histograms of image features can transparently move between ISR3 and Mneme.

Customizable data clustering. ISR4 offers more than storage support; it provides methods for customizing Mneme's buffer management and clustering policies according to an application's needs. For example, the database implementor can use Mneme's basic capabilities to introduce data clustering policies that reduce data access delays for specific applications, such as storing an image region and its computed features in the same physical segment. Similarly, features which are multi-dimensional in nature, such as geometric image structures, can be clustered on disk according to user-specified access patterns for efficient access.

An example of customizing storage and access for visual applications is the ISR4 *tile-*

image format, where an image is clustered on disk, and sub-images are brought into memory only as needed (on-demand). This ability significantly reduces the number of page swaps during common image processing operations.

Concurrent, Distributed Database Operations. Mneme supports concurrent database operations on arbitrarily complex objects within a distributed setting. It also provides customizable transaction and concurrency support, as well as extensible caching for use in client-server modes of operation.

Spatial and temporal indexing and query methods. ISR3 is equipped with a hierarchy of C++ classes that provide representations and methods for associatively and spatially organizing and accessing sets of memory-resident objects [3]. In particular, 2D geometric objects in images can be spatially stored into two-dimensional *grid*s [1] and retrieved according to spatial position in the image.

We are currently developing persistent versions of these access methods. When manipulating persistent data, these techniques can significantly reduce data access times because only the index data structures need to be kept in-memory when indexing persistent objects. Visual data reside on disk and are brought into memory only when accessed, *on-demand*. The access data structures are stored on disk at program termination for later use.

Mneme already provides one such standard indexing mechanism: the B+ tree. Moreover, Mneme provides the database programmer with a flexible and powerful interface for building different types of indices, including spatial indices, such as quad-trees and R-trees, and other multi-dimensional indices. A more general access structure similar to the Generalized Search Tree (GiST) [8] is also under construction for Mneme, which will be extensible in both the data types it can index and in the queries it can support.

We are also adding 3-D access mechanisms, and spatial and temporal query languages and techniques to this framework. A temporal index based on the Time Index [5], and optional versioning will also be provided to support historical queries. Once indices are built, query languages and techniques will also be implemented within this framework.

Extensibility. ISR4 offers generic solutions that lend themselves to immediate use by the visual database implementor, such as concurrent and distributed storage and retrieval ability for arbitrarily complex objects. However, a single generic solution is not suitable for more specific needs, such as application-dependent data structures, query methods, and indexing mechanisms. In such cases, ISR4 provides an initial set of powerful tools, and leaves it to the database implementor to generate representations, operators, indices, and query facilities tailored to the application. As an example, ISR4 allows—and encourages—the user to extend its initial set of representations by adding new ones. Visual data types can be easily defined and integrated with the system using ISR4's data definition language (DDL) (see Section 4.3.5). Likewise, Mneme is fully accessible for building multi-dimensional indices, or for tuning the buffer management and data clustering policies to the application-specific data requirements.

4.2 The Architecture of ISR4

ISR4 has two major components: a Database Programming Interface (DPI) based on ISR3, and a Storage Manager (SM) based on Mneme. The DPI provides a data definition language for defining visual objects, a hierarchy of visual representations, I/O support for visual data in a variety of common file formats, and graphics tools for displaying visual data. It also provides an initial set of indexing techniques for storing and retrieving (primarily 2D) data. The Storage Manager, on the other hand, provides concurrent and distributed storage, and customizable indexing, buffer management, and disk clustering support for persistent visual objects.

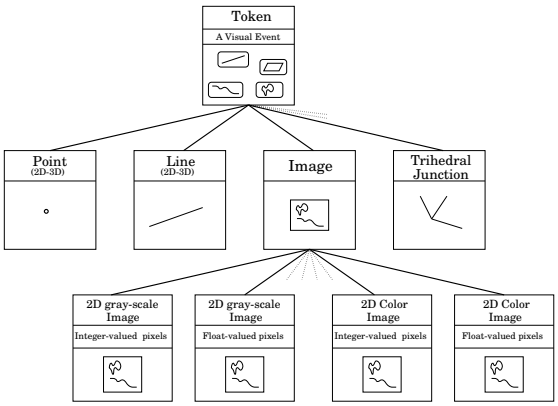

Figure 2: ISR4's Object Class Hierarchy includes representations for single-band and multi-spectral images, 2D points, lines, edges, and regions, and 3D points, lines, surfaces, and volumes.

4.3 The Database Programming Interface

4.3.1 ISR4 Object Class Hierarchy

The DPI first answers the representational needs of a VIMS with a hierarchy of representations for visual objects. It is essential for a VIMS to be equipped with a rich set of visual representations, especially if it is aimed to support a wide range of applications. As shown in Figure 2, the root of the ISR4 object hierarchy is the *token*—a basic abstraction for a visual object. A token[2] defines a visual object via its attributes, and input, output, and display methods. ISR4 provides an initial set of commonly used token classes, including (several types of) images, 2D and 3D points, lines, regions and edges. Although rich, the initial set of tokens may not be appropriate for all applications. Likewise, an originally appropriate representation may have to adapt to an evolving set of data requirements. The DPI addresses these issues by allowing the modification or extension of its data types. A Data Definition Language (DDL) is provided for defining new tokens, or modifying the existing ones, and which also ensures that a new token is transparently integrated with the rest of the system (see Section 4.3.5).

A similar object hierarchy is given as part of the Image Understanding Environment (IUE;[15]). IUE is an object-oriented environment that aims to facilitate exchange of research results within the Image Understanding (IU) community by providing the basic data structures and operations needed to implement IU algorithms.

ISR4 differs from IUE in that it emphasizes flexibility, simplicity, and extensibility, while IUE's focus is on completeness. ISR4's flexible object hierarchy allows researchers to add new objects, and define their own object relationship constraints. IUE's hierarchy, on the other hand, aims to provide a standard, complete class hierarchy, which specifies where any visual data type and process falls in the taxonomy. ISR4's hierarchy is kept simple to mainly focus on visual objects, in an attempt to offer uniform treatment of all its data types. All ISR4 objects, including complex objects (such as a set of image curves and spatially organized image features) are guaranteed input, output, persistent storage, and "display" support. Such uniform treatment is not possible in the complex IUE object hierarchy, which includes abstract mathematical objects, such as transforms and camera models, and procedural objects, such as tasks.

4.3.2 Associative and spatial access methods

The second DPI contribution comes in the form of representations and methods for associatively and spatially organizing and accessing sets of memory-resident objects. Visual objects, especially features computed in an image, are often loosely structured into sets or groups [1]. For example, in content-based query operations, related or multiply-occurring features frequently need to be treated as single entities, so that operations can be performed on

[2] A token is really an instance of a C++ class derived from Token, although we will refer to both the instance, and the type as a token whenever the difference is clear by the context.

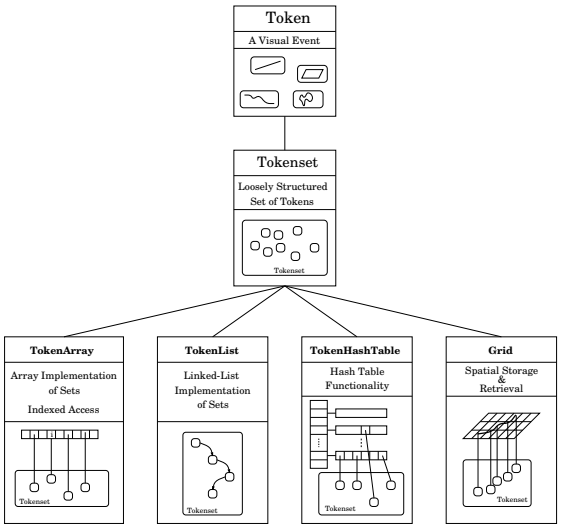

Figure 3: ISR4 provides a hierarchy of representations for associative and spatial token organization and access.

the group as a whole, as when an image-analyst executes a polygon-finder on the set of lines (the group of features) extracted from an aerial image in order to form building hypotheses.

At the other extreme are operations that require isolating elements in a group that satisfy certain desired properties, such as finding the image lines whose orientation lies within some specified range. As shown in Figure 3, the DPI serves both purposes with a hierarchy of C++ classes derived from token, called *tokenset*s. Set operations and range queries are defined on tokensets, as well as associative access methods. In particular, geometric objects can be spatially stored into two-dimensional *grid*s and retrieved according to spatial position in the image [1]. As before, ISR4 is extensible, so that other grouping techniques, such as oct-trees for 3D data can easily be implemented.

4.3.3 Flexible I/O for visual objects.

As a third form of assistance, the DPI provides input and output support for visual data produced in operations. In the DPI, any recognized token or set of tokens can be output to a file and read back. The I/O methods allow storing tokens in both ASCII and binary format files. The ASCII format file, *isra*, allows the inspection of tokens stored in it, while the binary format, *isrb*, is more compact and faster to read. In addition to ISR4's own formats, the I/O routines support reading and writing image files in many commonly used external formats.

4.3.4 Common image formats and graphics.

A set of common image formats comprise another form of support. Of all the visual data types that need to be managed in applications, images are the largest and most prevalent. Consequently, images need special attention in a VIMS. To address the needs of a wide variety of applications, the DPI supports most commercially available image formats, including gif, tiff, (Khoros [17]) viff, (KBVision [19]) im, and (soon) JPEG. This allows image processing and image understanding operators from such widely-distributed systems as Khoros and KBVision to be applied to ISR4 data. To support the smooth integration of any new operators with the system, the DPI provides facilities to convert any image format to another. External to the DPI, ISR4 supplies interactive graphics for displaying and inspecting most system data objects stored in files, with an executable called *xisrdisplay*.

4.3.5 ISR4 Data Definition Language (DDL)

One way to define data types (not provided in ISR4) is to introduce an external language for defining the characteristics of objects, and convert the descriptions with a preprocessing step to C++ class definitions recognized by ISR4. A second approach, which is favored here, is to directly define objects using C++ syntax. This eliminates the preprocessing step and the need to learn a separate language for object definition. While it is true that a DDL can be more powerful in expressing object-oriented

concepts than a programming language, the abstractions provided by C++ for object definition and manipulation are sufficient to satisfy the needs of typical VIMSs.

As mentioned earlier, a visual object is mapped to an instance of a C++ class called `Token` in ISR4. The Token type defines the global properties of objects and the available operations on them. All ISR4 objects derive from `Token` to inherit these properties. The definition of an ISR4 object type takes the following form:

class *ClassName* **:** **public** `Token`[3] {
  public :
  – Presentation support methods:
    `name(); trace(); draw();`
    `input(); output();`
  – Storage Manager support methods:
    `size(); traverse();`
    `pack(); unpack();`
  – The object's own public field and signature definitions
  protected :
  – Definitions visible only to subclasses of *ClassName*
  private :
  – Private field and signature definitions.
};

Objects of arbitrary complexity can be defined using this definition rule, as there is no restriction on the object attributes. A token must be introduced to the system via a set of support methods, however, so that it can properly be manipulated in the system. As listed above, the support methods that need to be supplied are `name`, `input`, `output`, `trace`, `draw`, `size`, `traverse`, `pack`, and `unpack`. These methods are very easy to generate; they require at most one line of code for each attribute of the token. The `traverse` method, for example, which visits the objects referenced by a token, consists of calls to a system function for every object reference it has:

```
class TrihedralJnct :  public Token {
 public:
  Line2D *Segment1;  // Segments
  Line2D *Segment2;  // forming the
  Line2D *Segment3;  // junction

  Point2D *Center;   // Junction point
};

TrihedralJnct::traverse(TravTable& tt)
{
   tt.add (Segment1); // Add the object
   tt.add (Segment2); // to the list
   tt.add (Segment3); // of objects
   tt.add (Center);   // traversed.
}
```

Once recognized by the system, an object can be: 1) Included in a tokenset, and accessed associatively or spatially; 2) Stored in and retrieved from the persistent object store; 3) Output to, and read back from a file; and 4) Displayed and inspected using *xisrdisplay*.

4.4 The Storage Manager

The Storage Manager, based on the Mneme persistent object store, provides storage support for complex visual objects. As described below (Section 4.4.2), the Storage Manager mainly acts as an interface that maps Database Programming Interface requests and objects to corresponding Mneme requests and objects. The actual storage and retrieval of objects, and the associated buffer and disk management is carried out by Mneme.

4.4.1 Mneme

The Mneme object store aims to provide the illusion of a large heap of objects, directly accessible from the Storage Manager. The main abstractions provided by Mneme to the Storage Manager are objects, object pointers, files, object pools, and buffer pools [13]. Mneme views an *object* to be a collection of bytes and references to other objects. Each object is uniquely referenced by an *object identifier*. In Mneme, objects are grouped together into units called *file*s. Each file has a special object called the *root* object, which can be used to store references to and information about the objects that

are stored in that file. Within files, objects are logically grouped in *object pool*s according to the policy under which they are managed. The policies governing the management of objects in a pool are dictated by a set of routines called *strategies*. Mneme defines a few pool strategies, although the users can supply their own specialized strategies. Each Mneme object is a member of exactly one pool, and each pool a member of exactly one file.

As mentioned above, Mneme currently supports the B+ tree index for indexing visual objects on disk, and a more general customizable access structure is under construction, which will support different types of application-specific indexing and queries in one structure.

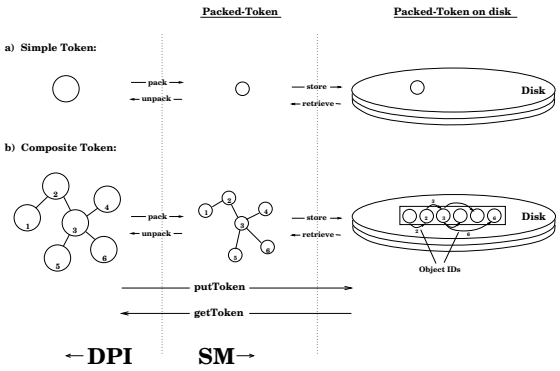

Figure 4: The Storage Manager provides interface functions between the DPI and Mneme.

4.4.2 Object Storage and Retrieval

As mentioned above, a token is defined in the Database Programming Interface using the ISR4 DDL. Typically, a token is born transient when created in the space of the DPI. When it needs to be made persistent, the DPI passes the token and its name (to be given) to the Storage Manager with a request to store it in a specific file. The storage request may include clustering parameters, such as "near object X." Alternatively, this can be done automatically by inserting the object into an index structure which will store the object in a way to access it rapidly. When multiple indices that have conflicting clustering requirements need to be kept on an object, multiple copies of the object can optionally be made when the object is immutable (known not to be modified after creation).

To save storage space and reduce database access time, ISR4 tokens are stored in Mneme in a compact format. Before an ISR4 token is stored, the Storage Manager maps it into a compressed Mneme object. Similarly, the Storage Manager retrieves a compressed object in Mneme, and uncompresses it before passing it to the DPI. As the `pack`, `unpack`, and `traverse` methods used during compression and decompression are required as part of a token's definition (see Section 4.3.5), storage and retrieval support is guaranteed for any ISR4 token, including complex objects such as a spatial grid of line segments or a histogram of image features.

The Storage Manager then acquires a pointer to a corresponding empty Mneme object in the desired file, and maps the token into the Mneme object, compressing it in the process. Mneme also assigns a unique id to the object. The Storage Manager accesses the object using its id or pointer.

In the DPI, on the other hand, access to the token is through its given name. Since names are not stored as part of an object in Mneme, the Storage Manager keeps a translation table (for each Mneme file) to convert the name to an id, and uses the id to access the object. When the identifier of a desired object is presented, Mneme returns a pointer to the memory-resident object. If the object was not previously memory resident, it is brought into memory from the file that contains the object. The Storage Manager then uncompresses the object into an ISR4 token, and passes the token to the DPI.

Storage and retrieval is more complicated when the object to be stored or retrieved has references to other objects (see 'composite object' in Figure 4). The object references are

converted to ids during storage, and back to direct pointers during retrieval. The performance analysis of this scheme called *copy swizzling*[4] has been thoroughly investigated in [12], and is beyond the scope of this paper.

The Storage Manager also provides methods to lock and release objects in Mneme buffers. Mneme guarantees to keep an object resident between the time when a pointer to the object is first obtained and the time when a release operation is performed on the object. Mneme can force a released object to disk to make space for another object. All memory-resident objects are transferred to secondary storage at program termination. To ensure that the name-to-id correspondences are valid during different executions of a program, the translation table is automatically stored in the corresponding Mneme file when a program terminates, and restored when the file is opened.

4.5 The Tile-Image Format

ISR4's *tile-image* storage and access model achieves improved storage and access performance by tuning the interaction between the storage manager's buffers and disk segments. The idea is to divide an image into (possibly overlapping) *tile*s, and represent the image by the collection of all the tiles. The tiles are stored as separate objects, and are brought into memory only when the image subregion covered by the tile is being accessed. Image tiling strategies are among the most straightforward, but there are many others. One example is mapping multiple dimensions (eg. 2D, temporal, function type of site models, etc.) into one dimension of disk clustering. Figure 5 shows a tile-image with overlap between tiles.

The tile-image model is based on the observation that only part of an image is accessed at a time in typical applications. In the RADIUS project, for example, it is common to divide

[4] Swizzling refers to the replacement of id references between memory-resident persistent objects with direct pointers. Copy swizzling is making a separate copy of the object being swizzled, versus carrying the swizzling *in-place*, in the object manager's buffers.

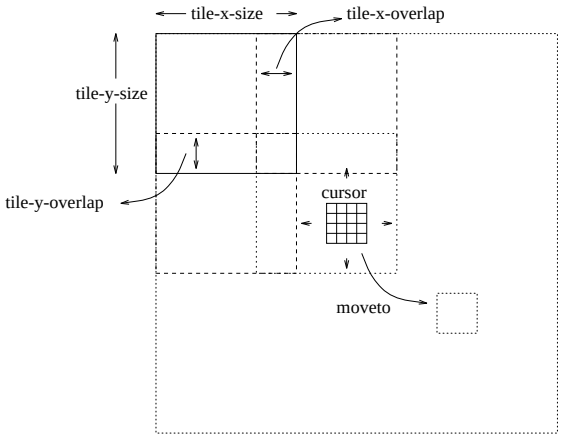

Figure 5: Tile-image with tile-overlap and Cursor.

an image and access its sub-images according to functional area in a site model. The tile-image model makes it possible to work on an image even though only parts of it are memory-resident. This is desirable especially when the whole image does not fit into physical memory, and only a sub-image can be read at a time. RADIUS images, for example, are typically 10K$\times$10K and larger. On the other hand, even when the memory is large enough to accommodate the entire image, we may want to limit the amount of buffer space we use for it, so that there is room left for other objects, such as the building models extracted. Typical applications that work with large images maintain a window into the image and keep in memory only the sub-image that falls in the window.

The model also aims to take advantage of the fact that images are rarely modified, except at creation time. This allows focusing on read-only operations when optimizing buffer management for image data. For example, the parts of the image read into memory can be accessed directly in Mneme's buffers (rather than creating a copy as with other tokens)—also called *in-place swizzling*—and a buffer page holding image data can be recycled without having to worry about writing its contents back to disk.

Once created, a tile-image can be accessed

either in a *random-access* mode, or a *scan* mode. The two access modes represent the two most typical access patterns found in visual applications. The random-access mode is for cases when pixels from distant image locations are accessed consecutively, resulting in a 'jump' from one location to the other. Such a pattern would arise, for example, when examining pixel values from different visual objects in the image. The scan mode, on the other hand, is for convolution-like image processing operations, where every pixel is visited one by one, and some value is computed based on the pixel and its neighbor pixels.

These access modes are provided via a *cursor*, which is a window into the image that allows access to the pixels located in the window. In the random-access mode, the cursor can be moved to any location in the image, whereas in the scan mode, it is allowed to move only locally (left, right, up, and down).

An issue related to images that still needs to be addressed is the storage and retrieval of compressed images. It is common practice to store large images in a compressed form. It is therefore desirable to support transparent access to images, where the user does not know whether the image is compressed in the database or not. The same problem arises when dealing with continuous media, such as video data. Video sequences are stored on disk in compressed form (e.g. MPEG;[7]), but have to be transparently uncompressed during playback. This can be achieved by presenting an image (or video frame) to the application program only after uncompressing it—uncompressing an image that is not compressed being a null operation. We may want to copy-swizzle compressed images, in this scheme, however, since uncompressing creates a (larger) copy of the object in any case.

5 Motivating Examples

5.1 Content-based Image Retrieval

A number of current application areas exist that would immediately benefit from using ISR4. One is content-based image retrieval, for example, the QBIC [6] project. In QBIC, color, texture, shape and sketch features are computed for image areas outlined by the user, and used at query time for image retrieval. The features, which consist of objects as complex as histograms and reduced resolution edge maps, are currently stored in an extensible relational database called Starburst [11]. The images themselves, on the other hand, are stored in flat files.

One can achieve better data clustering and faster data access if the images and related features are stored using the strategies of ISR4. First, ISR4 will directly support the storage of QBIC objects, so there is no need for disk-to-memory data format transformations, as in the current transformation from tuples to objects. Second, image features can be associated with the image region they came from and stored and retrieved with the sub-image. This is useful in QBIC, especially when one wants to see which features (if any) were selected from an image region. Accessing the region will retrieve the corresponding features as well, which can then be displayed. Feature indexing capability is also critical in QBIC. The current B+ tree index can be used for fast object retrieval, and different types of multi-dimensional indices can be built and incorporated into ISR4. Along with indices, query mechanisms can also be implemented.

5.2 Site Models for Photo-interpretation

Intelligence gathering operations provide other applications. As an example, the RADIUS project [16] is developing Image Understanding (IU) tools for image-analysts to support automated 3D cite model acquisition, model extension, and change detection. In a typical scenario, analysts build up a folder of image data and other intelligence about a site. Based on this information, analysts form a 2D map of the functional areas of the site, including abstract features such as the typical number of

cars found in each parking lot. Finally, 3D models of the permanent structures in each area are built. Once a site model has been developed, future images and intelligence reports can be compared to it in a set of processes called "change detection," in which analysts search for any temporal change in the functional areas, features, and/or structures in a site.

Here, 3D geometric site models plus collateral information, such as text, maps, and representative imagery, need to be stored in a fashion that allows efficient data retrieval for change detection programs, as well as interactive query support for photo-analysts and military planners. Currently, the RADIUS Testbed Database (RTDB)[9] stores complex objects such as geometric models, collateral data, and imagery information in a relational DBMS (Sybase), while image pixel data are stored in flat files.

ISR4 would allow RADIUS features to be grouped, stored, and retrieved according to their functional areas. Images would be partitioned according to functional areas, and the subimages would be clustered on disk with their associated features. In addition to fast access to image objects, this approach leads to better buffer management, especially with large aerial site images, since it restricts data movement to only a small, relevant portion of the image. Since RADIUS images are typically $10K \times 10K$ pixels or larger, such efficient buffering mechanisms are required. As with QBIC, spatial indices, as well as query languages, can be built using ISR4 to answer interactive queries from analysts and planners such as 'In the newly-arrived set of images, give me the ones in which there is a new structure,' or 'Give me the image (folder) of this site in which this building appears for the first time.' To support historical (time-based) queries, the functional areas can be linked over time to form a spatio-temporal sequence, over which site structures are indexed.

In a similar manner, ISR4 can support other applications with visual representations, operators, and storage management, including astronomy (sky survey) databases, geographic and environmental information management, CAD tools, and medical imaging.

6 Conclusion

Visual applications need to efficiently represent, manipulate, store, and retrieve both raw and processed persistent visual data. Extensible visual object stores offer effective means to address the data management needs of visual applications. ISR4 is an extensible visual object object store that will offer extensive storage and retrieval support for complex and large visual data, customizable buffering and clustering, and spatial and temporal indexing. In doing so, it will provide a variety of multi-dimensional access methods and query languages. Query optimization, along with approximate, ranked query methods, are among planned future additions.

References

[1] J. Brolio, B. Draper, R. Beveridge, and A. Hanson. ISR: A Database for Symbolic Processing in Computer Vision. *IEEE Computer*, 22(12):22–30, 1989.

[2] Editors: H. I. Christensen and J. L. Crowley. *Experimental Environments for Computer Vision and Image Processing*. World Scientific, 1994.

[3] B. A. Draper and G. Kutlu. *ISR3.1 User's Manual*, 1994.

[4] G. Droge and H.-J. Schek. Query-Adaptive Data Space Partitioning Using Variable-Sized Storage Clusters. In *Advances in Spatial Databases: Proceedings of the 3rd International Symposium SSD*, pages 337–356, 1993.

[5] R. Elmasri, G. Wuu, and Y. Kim. The Time Index: An Access Structure for Temporal Data. In *Proceedings of the Conference on Very Large DAtabases*, pages 328–336, August 1990.

[6] W. Niblack et. al. The QBIC Project: Querying Images By Content Using Color, Texture, and Shape. In *SPIE, Storage and Retrieval for Image and Video Databases*, volume 1908, pages 173–187, 1993.

[7] D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. Technical Report No. 1252, University of Wisconsin at Madison October 1994.

[9] A. Hoogs and B. Kniffin. The RADIUS Testbed Database: Issues and Design. In *IUW, Monterey, CA*, volume 1, pages 269–276, Nov. 1994.

[10] R. Jain. Workshop Report: NSF Workshop on Visual Information Management Systems. In *SPIE, Storage and Retrieval for Image and Video Databases*, volume 1908, pages 173–187, 1993.

[11] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to Starburst: Objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, 1991.

[12] J. E. B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Trans. on Software Engineering*, 18(8):657–673, 1992.

[13] J. E. B. Moss, T. Hosking, and E. Brown. *Mneme V3.x User's Guide*, 1994.

[14] J. Eliot B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[15] J. L. Mundy and the IUE Committee. The Image Understanding Environment: Overview. In *IUW, Washington, D.C.*, pages 238–288, April 1993.

[16] J. L. Mundy, R. Welty, L. Quam, T. Strat, W. Bremmer, M. Horwedel, D. Hackett, and A. Hoogs. The RADIUS Common Development Environment. In *IUW, San Diego, CA*, pages 215–228, Jan. 1992.

[17] J. Rasure and S. Kubica. The Khoros Application Development Environment. *In [2]*, pages 1–32, 1994.

[18] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *International Conference on Data Engineering*, volume 10, pages 328–336, 1994.

[19] T. Williams. Image Understanding Tools. In *ICPR, Atlantic City, N.J.*, number 10, pages 606–610, June 1990.