ISR3.1 User's Guide & Programmer's Reference

Bruce A. Draper   Gökhan Kutlu

March 30, 1995

Contents

List of Figures

List of Tables

1 Introduction

ISR3.1 is a software support tool that helps computer vision researchers store, retrieve and communicate visual data, with an emphasis on symbolic rather than image-like data. It was designed based on experience gained at the computer vision laboratory of the University of Massachusetts, where it was learned that: 1) vision systems typically create more symbolic data, in the form of points, lines, and similar structures, than iconic (image-like) data; 2) this symbolic data is often loosely structured into sets or groups; and 3) the ability to efficiently store and access this data is critical to the success of computer vision systems [3]. ISR3.1 therefore supports vision researchers by providing an in-memory database for C++ class instances with routines for rapid associative and spatial retrieval.

Another common cause of failure for computer vision projects is systems integration. Many computer vision systems are supposed to be assembled from semi-independent software modules developed by different researchers, sometimes from different laboratories. Unfortunately, integrating these pieces into a single coherent system invariably proves to be a complex and time-consuming process, and sometimes the practical difficulties exhaust the resources of the project. Mant of these systems integration problems can be traced to 1) incompatible structures for symbolic image data or 2) incompatible storage (typically file) formats. When the integrated system is supposed to operate in a real-time environment, interprocess communication (IPC) becomes a third source of problems. ISR3.1 supports (computer vision) system integration by supplying a default library of common visual object classes, such as points, lines and regions. It also provides file I/O routines to ensure that such data is written and read uniformly (in either ASCII or binary format), and for real-time applications an IPC stream mechanism that complements the current file I/O capabilities is being developed. Recognizing that many useful vision software modules have already been developed that do not use ISR3.1 classes or I/O routines, ISR3.1's I/O routines also support reading and writing files in many commonly-used formats, including *gif, tiff, viff* and *im*.

Realistically, however, ISR3.1 is not a software panacea. It cannot turn poorly written code into efficient and portable software modules, nor can the ISR3.1 library anticipate every data structure that a computer vision researcher might need. Instead, ISR3.1 is an object-oriented (OO) system that we hope will encourage vision researchers to develop object-oriented code. Like all object-oriented systems, the emphasis in ISR3.1 is on software reuse and system extensibility. We have written data storage and retrieval routines, I/O functions and the rest so that other vision researcher's won't have to. If ISR3.1 lacks an object class or functionality that a researcher needs, its object-oriented structure is designed to allow it to be easily extended. This will eventually lead to some degree of incompatibility between version of ISR3, but if researchers who extend the system make their code available, we can all benefit from each other's labors.

1.1 Input/Output (I/O)

The core of ISR3.1 is a library of C++ classes defining symbolic computer vision data structures, such as points, lines and faces. These classes are called *token types* in ISR3.1, and instances of these classes are called *tokens*. Each token contains data describing one visual event, such as a point or line in the image. In addition, all tokens can be written to and read from files, or stored in sets to be retrieved later; some tokens can also be graphically displayed. Fundamental to the design of ISR3.1 is the notion that new token types can and should be introduced by the user. Although ISR3.1 comes with an initial set of token types, including tokens for images, points, regions and lines, there has been no attempt to forecast all the needs of vision researchers. Instead, ISR3.1 is designed to be an extensible system that will grow as users define new token types.

For file I/O, ISR3.1 defines an output stream class called TokenOStream and an input stream class called TokenIStream. Any token can be written to a TokenOStream or read from a TokenIStream. If a token contains pointers to other tokens (for example, a pair token that contains pointers to two line tokens), ISR3.1 will not only write out the initial token, it will also recursively write out any token it has a pointer to. When the token is read later on, the dependent tokens will also be read in and the pointers will be restored. This capability will work even for circularly linked tokens.

In addition, the file format of a token stream can be set at run-time. ISR3.1 has two native file formats: *isra* is an ASCII format that is designed to be easily read by humans, whereas *isrb* is a more compact binary format. Typically, tokens will be stored in binary (isrb) format to save disk space, but these binary files can be converted to ASCII (isra) any time a person wants to look at their contents. Token streams can also be set to various foriegn file formats, such *gif* (used in Mosaic and on many PCs), *tiff* (used on Apple MacIntoshes), *viff* (used by Khoros [2]) and *im* and *tks* (used in KBVision [1]). More foriegn file formats, including *jpeg* and the *IUE data exchange format* are planned.

When development is complete, users often want to embed vision modules in larger systems by using the output of one module as the input to another. At this stage, File I/O is no longer appropriate: data should be passed directly from one module to another. An interprocess communication (IPC) module is currently under development for ISR3.1 that will let token streams be bound to other processes (using the UNIX stream mechanism). This will allow data to be passed directly from one process to another.

1.2 Token Sets

The most important capability of ISR3.1 is the ability to store tokens in sets, and then retrieve individual tokens or subset of tokens from those sets. ISR3.1 actually supports many implementations of sets: token sets can be arrays, lists, hash tables or 2D grids. However, all of these classes of sets are derived from the *TokenSet* class, and all provide the same basic storage and retrieval functions, so for many applications it does not matter which set implementation is used.

The basic operations on sets are the logical ones; tokens can be added or removed from sets, or tested for membership in a set; and programs can take the union, intersection or difference of two sets. Functions are provided for iterating through tokensets, or sets of tokens can be retrieved by their feature values. Special types of token sets may provide additional retrieval mechanisms; Grids, for example, allow tokens to retrieved by their 2D spatial location. TokenSets are typically used to store a set of closely related tokens, such as the set of line segments extracted from an image. Unlike in previous versions of ISR, however, it is possible to store several different types of tokens within a single token set.

1.3 Utilities

In addition to the class library, ISR3.1 comes with a set of executable utility programs. Most of these are simple programs, such as one that converts a system file from any format known to ISR3.1 to any other. One complex utility, however, is *xisrdisplay*, a graphics program for displaying tokens. Xisrdisplay can be used to graphically display the contents of a file or, by using token streams as an IPC mechanism, to show intermediate results of a program as it is running.

2 Tokens

ISR3.1 provides storage, retrieval, I/O and graphics capabilities for *tokens*, where a token is an instance of a C++ class that inherits directly or indirectly from the class **Token**[1]. In general, storage and retrieval facilities are provided by the **TokenSet** and **Grid** objects; tokens can be stored in tokensets and later retrieved by their attribute values, or by iterating through the tokenset. Tokens stored in Grids can be retrieved by their spatial location. Although TokenSets and Grids are sufficient to support the storage and retrieval needs of most computer vision algorithms, users are welcome to develop new types of TokenSets (based on B-trees, or Oct-trees, or whatever) if they so desire.

2.1 Simple Tokens

In order for a C++ class to qualify as a token class, it must inherit from Token and must provide a definition for the virtual function **name().** Name is a simple function that returns a string which is the name of the class; its primary function is to determine class equality. Two instances are assumed to be of the same class if their name() methods return the same string.

As an example of what a token class definition might look like, Figure 1 shows the definition for a class of image line segments called **Line2D.** This is a simplified version of the actual Line2D definition provided as one of ISR3.1's initial classes; the full definition will be shown later in section **??**. Figure 1 does show the name() method, however, and as such is a sufficient definition to allow Line2D instances to be stored into and retrieved from TokenSets and Grids.

[1]Inheriting from Token is a necessary but not sufficient condition for being an ISR3.1 token. The next section, Section 2.1, will present the minimal set of methods required of a token, and then future sections will present additional methods that, if defined, increase the power of a token.

```
class Line2D : public Token {
 public:
  float x1;
  float y1;
  float x2;
  float y2;
  float theta;
  float contrast;
  float dispersion;
  float length;

  char *name() { return "Line2D"; }
};
```

Figure 1: A simplified version of the Line2D token definition showing the name method definition.

The second requirement for making a class into an ISR3.1 token class is to list the class name and source (.h) file in the system's token file. By default, the token file supplied with ISR3.1 is called *ISRtoken*, and new tokens should be added to it. (Applications can also have their own token files, but that will not be described here.) Each line of the system's token file should contain the name of one token class and the source file it can be found in, as shown in Figure 2. If multiple token classes are defined in a single file, each class should be given its own entry.

2.2 Tokens with Multiple Inheritance

The following section should ONLY be read by users who NEED to define tokens with multiple inheritance. Most token classes should not need multiple inheritance, which is supported but discouraged by C++.

Most token classes inherit from a single parent class, which may be either the *Token* class or a previously defined subclass of *Token*. In some cases, however, a token designer may want to take advantage of multiple inheritance, in which a token class derives from two or more parent classes. In

```
BytePlane            NewPlane.h
IntPlane             NewPlane.h
FloatPlane           NewPlane.h
Line2D               Line2D.h
Line2DIntersection   LinePairs.h
Line2DPair           LinePairs.h
UV_transform         LinePairs.h
TokenArray           Container.h
TokenList            Container.h
Edgel                Edgel.h
```

Figure 2: Part of the default ISRtokens file supplied with ISR3.1. New tokens should be added one to a line, with each line containing a class name and corresponding source (.h) file.

this case, each token class must inherit from *Token* exactly once, and that inheritance cannot be via virtual base class inheritance. To understand this restriction requires a brief detour into multiple inheritance mechanisms in C++.

2.2.1 Multiple Inheritance

Many object-oriented programming languages, including C++, allow for multiple inheritance, in which a class is derived from two or more parent classes. In general, multiple inheritance works like replicated single inheritance, in that every field and method of a parent class is inherited by the child, unless the child redefines the field or method. Classes with multiple parents simply inherit from more than one source.

Multiple inheritance gets tricky, however, when conflicts arise. If more than one parent defines the same method or field, which one does the child inherit? In C++, the child inherits a complete copy of all parents, regardless of conflicts. If a method *foo()* is inherited from two different parents A and B, the child inherits both, with the first being referenced as *A::foo()* and the second as *B::foo()*.

The scheme of inheritance as concatenation works well as long as none of the parents share a common ancestor. Otherwise, instances of the child class will include multiple instances of the common ancestor class, as shown in Figure **??**. This is acceptable (although rarely desirable) in some applications, but not in applications such as ISR3.1 which rely on generic operations. In particular, ISR3.1 provides many functions that operate on or return pointers to tokens. As long as a token inherits only one copy of the *Token* class, the C++ compiler knows how to promote subclasses up the hierarchy to class *Token*, so any of these operations can be applied. If the subclass inherits multiple instances of the class *Token*, however, it is ambiguous which instance of Token the compiler should promote the subclass to, and a compile-time error results. Similarly, if an ISR3.1 operation returns a pointer to a token and the user's code tries to cast it to a pointer of a subclass that inherits *Token* from more than one parent, that too is a compile-time error. Hence the restriction that *each token class must inherit from* Token *exactly once*[2].

2.2.2 Virtual Base Class Inheritance

Recognizing that inheriting multiple copies of an ancestor class can be problematic, the designers of C++ added a second inheritance mechanism called virtual base class inheritance, in which an instance inherits at most one copy of any ancestor. If class A is a virtual base class of B, then instances of B actually inherit pointers to instances of A. Pointers are combined, so that an instance that inherits A indirectly from two or more parents will inherit multiple pointers to the same instance of A (although this indirection is hidden from the user). As a result, using virtual base class inheritance no instance will get multiple copies of an ancestor class.

Unfortunatly, this only solves half the problem. If a class has multiple (virtual) parents that (virtually) inherit *Token*, then ISR3.1 functions that operate on pointers to tokens can be applied to that class. Unfortunately, the user is stuck when these operations return a pointer to *Token*, because the virtual pointers are one-way; C++ will not allow you to cast a pointer to *Token* to be a pointer to a subclass that inherited it virtually. Another compile-time error results.[3] This leads to the second restriction that *ISR3.1*

[2] An alternative design choice would have been to have these operators take and return pointers to void. This would have short-circuited all of C++'s type checking mechanisms, however, and prevented these operators from being methods of *Token*.

[3] This makes sense. The subclass that virtually inherits *Token* contains a pointer to an instance of *Token*, but that instance does not contain a pointer back to the subclass

classes cannot inherit Token *as a virtual base class.*

3 TokenSets – Storage, Access and Retrieval

Although the ISR is centered around the *token*, in practice, tokens exist in aggregates and operations are performed on sets of tokens as a whole rather than on individual tokens in isolation. Many times, features extracted from images that are related, or multiple occurences of features—e.g. lines extracted from an image—need to be grouped or structured, to be treated as single entities, the same operation performed on the whole set, or elements retrieved in the set that have certain properties. Examples can be given such as **1)** storing lines belonging to the same object in a structure and having this structure in the system as a single item, **2)** filtering operations such as **i)** accessing all the lines belonging to an object that have a certain orientation, or **ii)** selecting all the lines which have length greater than 5 pixels (significant lines), **3)** grouping objects that come from the same region in the image and writing them in a file, **4)** intersecting lines belonging to two regions to obtain the set of lines that lie in both.

In most computer vision applications, 1) features would be extracted from an image and stored in some data structure. Other features of the same kind would be stored in other such structures, and 2) another data structure—typically an array or a linked list—would be the next place to store all features of the same kind. Then the user would probably be interested in 3) performing some operations on the features extracted, especially of the kind listed in the first paragraph in this section. Experience in the Computer Vision group at the University of Massachussets has shown that all these steps, especially the third one demands the investment of major efforts in the design and coding of flexible data structures and mechanisms to perform efficient operations on these data stuctures. The *token*, discussed in Section 2, serves as the general data structure in the first item here. The ISR3.1 *TokenSet*, described in this section, is designed to address the issues mentioned in the second and third items.

Let us first have a look at the ISR3.1 class hierarchy shown in Figure 3. *TokenSet* inherits from *Token*, and *TokenArray* and *TokenList* inherit from

(thats not part of the definition of *Token*). Hence, there is no way the system can trace back and convert a pointer to *Token* to a pointer to the subclass.

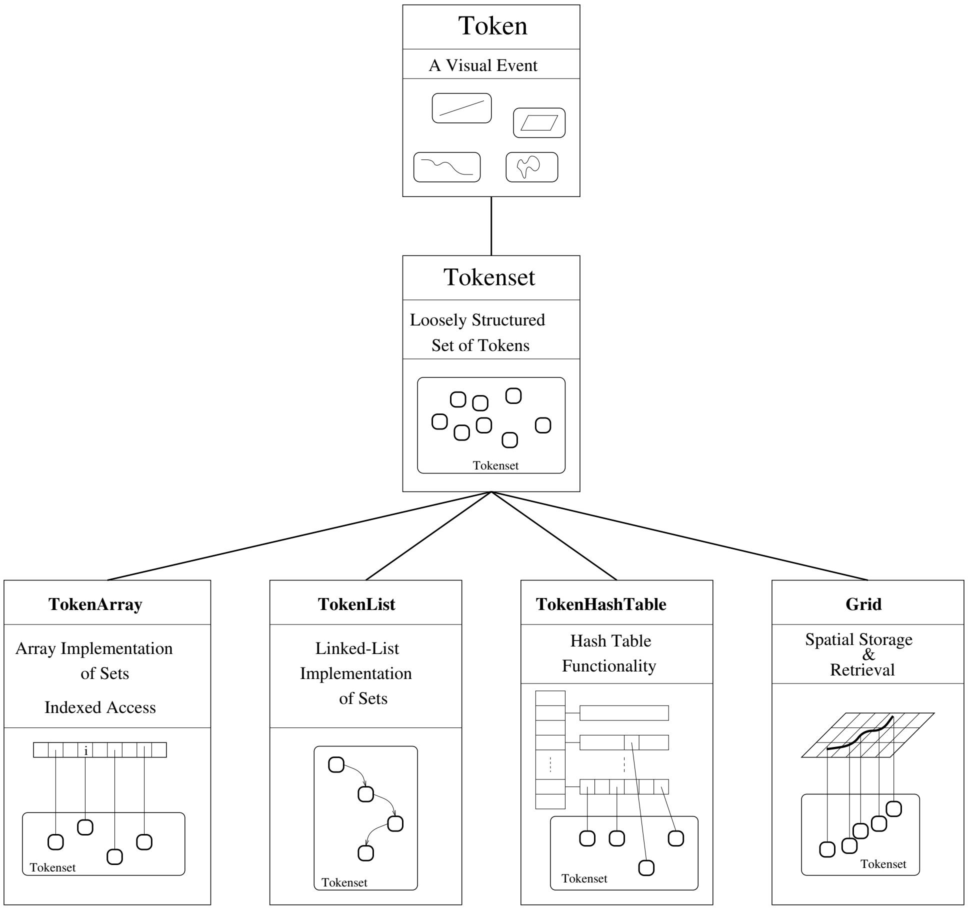

Figure 3: ISR3 Object Class Hierarchy.

TokenSet. *TokenArray* and *TokenList* share the properties of being a 'set of tokens'—i.e. a logical/abstract grouping of tokens—and a (physical) 'container' for tokens. Each of these classes and the methods they support is described later in section 8.

The *TokenSet* comprises of an in-memory storage structure and the set of methods[4] defined on this stucture. To make it simpler; a TokenSet is a place where tokens can be stored and accessed later. ISR3.1 TokenSets come with functions that can be used to *add* tokens to TokenSets, *remove* tokens,

[4]The terms function and method are used interchangeably when referring to functions defined within a class

access tokens sequentially, perform the set operations of *union*, *intersection*, *difference*, *select* and *retrieve* a subset of a set of tokens that satisfy certain conditions. There is no type restriction on the tokens that are stored in a TokenSet. Tokens of any kind can be kept in the same TokenSet (unlike in ISR3), therefore, in case retrieval or set functions are to be invoked for a TokenSet storing mixed type tokens, it is good practice to specify the type of token the retrieval is performed on (see section on retrieval functions). A *Transform* method performs a transformation on all the tokens stored in a TokenSet while *Draw* displays the contents of a TokenSet graphically.

A segment of ISR3.1 code which demonstrates the usage of the *add* method can be seen in Figure 4. This example also illustrates how objects of type TokenSet exist. **c2** is an object of type *TokenArray.* It is converted to an object of type **TokenSet** when passed as an argument to the Union method, since this class inherits from TokenSet. Although the *add* method is defined virtually for TokenSets, its usage is valid here, since this method is instantiated by an actual function in the definition of the *TokenArray* class (see Section 8.4.4). The description of the *TokenSetState* class is given in the next section.

The *retrieve* method is demonstrated in Figure 5, which is a useful function when tokens in a TokenSet that satisfy certain conditions need to be accessed. In this example, the requirement is that the lines retrieved have length in the range $16.0 \leq length \leq 22.0$.

The storage mechanism of TokenSets is implemented using an array (*TokenArray*) or list (*TokenList*) structure, as explained in Section 8.4.4 and Section 8.4.6 respectively. The array implementation is more suitable for cases where tokens are frequently accessed by their indeces, and removals are relatively few, whereas the list implementation is more efficient if tokens need to be inserted and removed often, and it is common to have searches visiting all the elements in a set to perform a certain operation. Although the underlying implementation is transparent to the user and the same methods can be used regardless of whether the object in hand is a *TokenArray* or *TokenList*, the user would highly benefit if the structural implementation fits the nature of the application.

```
/*-------------------------------------------------------------
  TokenSet& TokenSet::Union( TokenSet *c2)
  - Compute  this = this UNION c2;
  - Insert every token in c2 into 'this' if not already in it.
 -------------------------------------------------------------*/
TokenSet& TokenSet::Union( TokenSet *c2 )
{
  Token *token;
  TokenSetState *cs2;

  cs2 = c2->state();
  /* cycle through all the valid tokens: */
  for (token = cs2->value(); token != NULL; token = cs2->next())
    add( token ); /* add() insert()s into the TokenSet only if
                     'token' is not already there */
  return *this;
} // TokenSet::Union( TokenSet * )

main() {
  TokenArray *ta;
  TokenList  *tl;

  ta = new TokenArray;
     ..
  tl->Union( ta );
     ..
}
```

Figure 4: The *Union* method takes the union of *this* and *c2*. This example also demonstrates the usage of the *add* function and TokenSetState. For the user unfamiliar with the *this* pointer: the *this* pointer points to the object for which a method is invoked, **tl** in this case.

```
main() {
  Isristream isr_in;
  Isrostream isr_out;
  TokenArray *ta1, *ta2;
  int        type;

  /* Find out the type of the file to open from its extension. */
  match_stream_to_filename(isr_in, argv[1]);
  isr_in1.open( argv[1], ios::in); // open file
  if (!isr_in1) {
    cerr << "Unable to open file " <<  argv[1] << endl;
    exit(-1);
  }
  /* Find out the type of the file to open from its extension. */
  match_stream_to_filename(isr_out, argv[3]);
  isr_out.open( argv[3], ios::out); // open file
  if (!isr_out) {
    cerr << "Unable to open file " <<  argv[3] << endl;
    exit(-1);
  }

  /* Read tokens from ISR input stream isr_in into ta1. */
  ta1  = (TokenArray *) read_token_ptr(isr_in);

  type = token_index( "Line2D" );
  /* Find all lines whose length is in the range (16.0, 22.0). */
  ta2 = (TokenArray *)
          ta1->retrieve( type, OFFSET(Line2D, length), 16.0, 22.0);

  isr_out << ta2;
  isr_out.close();
}
```

Figure 5: The *retrieve* function in practice. Lines are read from a file and the ones which satisfy the condition $16.0 \leq line{\rightarrow}length \leq 22.0$ are retrieved and stored in another file.

3.1 TokenSetState

A structure that is used in conjunction with TokenSets is the *TokenSetState*:

```
class TokenSetState {
  public:
   virtual Token* value();
   virtual Token* next();
};
```

The user who wants to access the tokens in a TokenSet sequentially has to know how to use the *TokenSetState* structure. *TokenSetState*s are objects that exist only in relation to a TokenSet object. They are "one-directional" indeces into TokenSets, useful for a single scan of the tokens in a container. The following is an example of how to use the TokenSetState structure :

```
TokenSetState *cs;
TokenSet [5] *ct;
Token *token;

cs = ct->state();
for(token = cs->value(); token != NULL; token = cs->next())
    Perform some operation on token;
```

The **TokenSet::state** function returns a *TokenSetState* pointer initially pointing to the token that is passed as an argument, if any, or to the first token in the TokenSet. The **TokenSetState::next** function returns the next valid token if such a token exists and NULL otherwise. This is a useful feature, especially in cases when the tokens are not contiguously located in the container—as when a token is removed from an array, and the position that token previously occupied is empty and a gap is created. The **TokenSetState::value** function returns a pointer to the current token the TokenSetState "contains".
*TokenArrayState*s and *TokenListState*s, the instantiations of TokenSetStates, are described in Sections 8.4.5 and 8.4.7 respectively.

[5] See note at the end of Section 8.4.

4 Grids

2- and 3-dimensional data such as lines, regions, and surfaces are commonly used in Computer Vision. Many applications require frequent storage and access of sets of multi-dimensional structures. The search for items in the set satisfying certain spatial requirements is an expensive task that often becomes a bottleneck and seriously affects overall performance of vision systems. A solution to this problem is to store data in a way that reflects the spatial relationships among the data items in the set. Knowledge about the underlying storage structure can then be taken advantage of in order to access and retrieve stored data according to the requirements.

The **Grid** is the 2-dimensional storage and retrieval structure in ISR3.1, representing spatially-ordered sets of tokens. A Grid is a 2D array of "cells" which overlays an image, with each cell containing a set of tokens. The idea is to partition a region of interest into uniform-sized grids, and assign a storage bucket for each grid. Objects that are falling in the same grid are stored in the same storage bucket. An object that spans grid boundaries is stored in multiple buckets. The duplication is kept at the pointer level, by allowing storage of pointers to objects only. A bucket in ISR3.1 is called a **Cell** and the data structure that represents a Cell is the **CellElement**.

The basic operations defined on a grid are to store a token in a set of cells and to retrieve tokens from a set of cells. The only way to store a token in a cell is to activate the desired cell and call the function **Grid::add**. **activateCell** and **deactivateCell** are used to activate and deactivate cells. Typically, the cells that a token falls into are activated and the token is added into these cells. After this sequence of operations is performed for a set of tokens, the Grid is deactivated and specific cells that we are interested in are activated for the retrieval of the tokens stored in them. Then the **retrieve** function is called to access the tokens in the activated cells.

In the example in Figure 6, a grid is deactivated, and cells within a box around *line* are marked as ACTIVE by the function *rasterizePolygon* (see Section 8.5.3). Then the part of the lines that are outside the box are clipped and the ones that have orientation close to *line* are filtered.

Rasterization routines are supplied for selecting the cells that intersect a point, circle, line, scan-line rectangle or convex polygon.

These rasterization functions can be used either before or after a token is added to a grid. Before: to store a token into marked cells; After: to

```
Grid *lgrid;
Line2D *line;
COORDINATE box[4];
TokenArray *lines, *inBoxCoarse, *inBoxFine, *sameTheta;

lgrid->gridifyLines(lines); /* lines is a TokenArray of Line2D
                               objects */
...
/* A box is formed around 'line'. The code is omitted here. */
...
lgrid->deactivateGrid();
lgrid->rasterizePolygon(box, 4);

inBoxCoarse = lgrid->retrieve();
if (inBoxCoarse == NULL) continue;

/* Find the lines which are completely inside the box, and clip the
   ones which pass through the box, so that the resulting lines lie
   completely inside the box: */
inBoxFine = ISRclipLinesInBox(box, inBoxCoarse);

sameTheta = ISRorientationFilter(inBoxFine, pLine->theta, dt );
 ...
```

Figure 6: The lines around *line* within a box are retrieved. **deactivate** deactivates all cells in *lgrid*, **rasterizePolygon** marks all cells within *box* as ACTIVE and **retrieve** returns the tokens stored in the cells that fall in *box*. Another function, **ISRorientationFilter**, restricts the retrieved lines to lines which have orientations close to *line*.

```
/*****************************************************************
 * Grid::gridifyLines
 *  Store every token in a TokenSet of lines into the grid.
 *****************************************************************/
int Grid::gridifyLines( TokenSet *ts )
{
  Line2D   *line;
  TokenSetState *cs = ts->state();
  Token *token;

  /* For each line, store the token into activated grid cells */

  for(token = cs->value(); token != NULL; token = cs->next()) {
      line = (Line2D *) token;
      deactivate();

      /* Activate all grid cells which the line passes through */
      traceLine(  line->x1, line->y1, line->x2, line->y2, ACTIVE );

      /* Store line token into all activated grid cells */
      add( line );
  }
  ...
}
```

Figure 7: *Grid::gridifyLines* stores each line in *ts* into the cells that the line passes through. This is done by activating the cells that the line crosses by a call to *traceLine* and *add*ing the line into the activated cells.

mark (activate) cells in which tokens are stored that we desire to retrieve. For example, to store a set of line tokens in a grid, the function in Figure 7 (provided as a method in ISR3.1) marks cells as active and tokens are stored in these ACTIVE cells. And then, as in the example in Figure 8, the **Grid::rasterizeRectangle** method marks (activates) the cells for token retrieval.

As with TokenSets, the contents of a Grid can be output to a file with the ¡¡ operator as in the example in Figure 8. Here, Line2D tokens are read from a file into a TokenSet and then stored in *gr* with a call to *Grid::gridifyLines*. Then all the lines which lie in the rectangle (2.0, 3.0) (13.0, 27.4) are activated (**Grid::rasterizeRectangle**), retrieved (**Grid::retrieve**) and output to another file.

5 File I/O

One of the basic capabilities provided for tokens in ISR3.1 is file I/O. In particular, ISR3.1 provides functions for writing tokens to files in either an ASCII format or a more compact binary one, and for reading tokens from files of either format. In addition, for specific token types, the ability to read and write tokens in "foreign" formats, such as viff, im or tks, is also provided. (The file I/O functions discussed here are also compatible with the stream-based interprocess communication (IPC) facilities currently under development.)

The basic paradigm for using ISR's I/O functions is that an application program will open a file and either write or read a single token. ISR's functions will write (or read) the indicated token, plus any token it contains a pointer to, and any tokens they contain pointers to, etc. As a result, it is easy to write or read an entire tree of data with one command, and all pointers between tokens are preserved[8]. A typical file might therefore contain a TokenSet of line segments (or points, or surfaces). Such a file is written by writing the TokenSet; the line segments will be written automatically because the TokenSet contains pointers to them. Similarly, the TokenSet can be read by a single command that returns a pointer to the TokenSet, with the individual line segments being read as a side-effect. Note that if the tokens contain circular pointers, either direct (A has a pointer to B which has a pointer to A, or A has a pointer to itself) or indirect (A points to B points to C ... points to A), ISR3.1 will still write and read the pointers correctly, without getting into an infinite loop.

[8]Note that the semantics of pointers are preserved, not their values. If token A has a pointer to token B, then if they are written to file and read back in, A will still point to B. However, both A and B will have new locations in memory, so the value of the pointer will be different.

```
  TokenIStream isr_in;
  TokenOStream isr_out;
  TokenArray *ta;
  TokenSet *ts;
  Grid *gr = new Grid(-1, 32, 32); /* Declare a Grid of size 32x32, with
                                      the other parameters set to default
                                      values.*/
  isr_in.MatchFormatToFilename(argv[1]);
  isr_in.open( argv[1] ); /* Open a TokenIStream. */
  if (!isr_in) {
    cerr << "Unable to open file " <<  argv[1] << endl;
    exit(-1);
  }

  ts = (TokenSet *) isr_in.read_token_ptr(); /* Input tokens into
                                                TokenSet. */
  isr_in.close(); /* Close input stream. */

  gr->gridifyLines( ts );  /* Store lines in Grid. */
  gr->deactivate();        /* Deactivate the whole grid. */
  gr->rasterizeRectangle( 2.0, 3.0, 13.0, 27.4 ); /* Mark cells within
                                                     rectangle. */
  ta = (TokenArray *) gr->retrieve(); /* Retrieve tokens in marked cells.*/

  isr_out.MatchFormatToFilename(argv[2]);
  isr_out.open( argv[2] ); /* Open a TokenOStream. */
  if (!isr_out) {
    cerr << "Unable to open file " <<  argv[2] << endl;
    exit(-1);
  }

  isr_out << gr; /* Output contents of Grid. */
  isr_out.close(); /* Close output stream. */
}
```

Figure 8: Tokens are read in from a file and stored in *gr*. The ones lying within a desired rectangle are marked with a call to **Grid::rasterizeRectangle**, retrieved and output to another file.

5.1 Application-level I/O

To use ISR3.1's I/O facilities in an application program, the program must declare a **TokenIStream** (for input) or a **TokenOStream** (for output). Tokenstreams are streams to which only tokens can be written, and which have two properties, a *format* and a *filename*. In applications which do not use foreign file formats (see Section 5.3), the only formats a file will have are ISR3.1's ascii file format **isra** and its more compact binary format **isrb**. As with standard C streams, the filename specifies what file a stream is bound to.

When declaring a tokenstream, an application program may specify the format and/or filename, but is not required to specify either. If a filename is not specified when a stream is declared, one must be specified by the **open(char *)** method before the stream can be written to (or read from). A stream may be reused within an application by closing it and then re-opening it with a different filename (or the same filename, if the point is to overwrite the original file). If the format is not specified when a stream is declared it defaults to **isrb**, but the format can be changed using the **SetFormat(FileFormat)** method. The user should be wary of changing a file format once reading or writing has begun, however, as this will almost certainly lead to errors!

In general, the recommended way to use tokenstreams is to supply neither the format nor the filename in the declaration. Instead, the filename should be supplied by the user, generally in the form of a command line argument. The stream format can then be set using the **MatchFormat-ToFilename(char *)** method of token streams, which sets the format of the stream to match the filename extension: ascii if the filename ends in .isra, binary if it ends in .isrb. This method also recognizes many foreign file extensions, such as .plane, .viff and .im (see Section 5.3), so it creates code that can be applied to a wide variety of file formats.

Once a TokenOStream has been declared (and opened, if no filename was initially provided), tokens can be written to it with the standard << operator. For TokenIStreams, tokens can be read two ways: either by creating a token instance and filling it using the standard >> operator, or more commonly by declaring a pointer to a token and using the **read_token_ptr()** method of TokenIStream. Read_token_ptr() returns **(Token *)**, so its return value must be cast to the appropriate token type.

A good example of using token streams correctly can be seen in Figure 9. The ISRconvert function simply reads in a data file and writes it back out again – but since the stream formats are set to match the filename arguments, it can be used to convert files from ascii to binary or from binary to ascii. (This amazingly useful, albeit simple, utility is supplied as a executable file in ISR3.1.)

5.2 Defining Token Methods for I/O

Note: readers who are not interested in designing new token types may skip this section.

Unlike many other visual representation systems, ISR3.1 encourages its users to create new token classes to express new types of visual data. This can be done by adding new features to existing token classes (creating subclasses) or by starting from scratch and defining entirely new token types. In either case, three virtual functions must be defined in order for ISR3.1 to read and write the new token type. The first function, **trace()**, is used to determine how tokens are connected; it returns void after calling the function **Register(token *)** on every token instance that the current token instance has a pointer to. The default definition for trace is a no-op, so token classes that do not contain pointers to any other tokens do not need to redefine trace.

The **output(Isrostream&)** method specifies how instances of a token class should be written. It returns void, but take as an argument an **Isrostream**. Isrostreams (and Isristreams, their input counterpart) are the objects that write (read) tokens in ISR3.1's native data formats; they should not be manipulated at the level of applications programs (use TokenOStream and TokenIStream for this purpose; see Section 5.1). In most ways, Isrostreams are like standard C++ ostreams[9], except that Isrostreams have a format field that specifies whether the stream is ASCII or BINARY. In ASCII format, all numbers are written as sequences of ASCII digits, whereas in BINARY format all numbers are written in binary. The other major difference between Isrostreams and standard C++ ostreams is how strings are written and read. When a character string is sent to an Isrostream, it checks

[9] Isrostreams ought to be subclasses of ostreams, so that all valid operations on ostreams could be applied to Isrostreams as well. Unfortunately, a bug in the GNU C++ compiler (version 2.6) seems to prevent this.

```
main(int argc, char *argv[])
{
  TokenIStream isr_in;
  TokenOStream isr_out;
  Token *data;

  if (argc != 3) {
    cout << "Usage: ISRconvert filename1 filename2\n";
    return 0;
  }

  isr_in.MatchFormatToFilename(argv[1]);
  isr_in.open(argv[1]);
  if (!isr_in) {
    cerr << "Unable to open file " <<  argv[1] << endl;
    exit(-1);
  }

  data = isr_in.read_token_ptr();

  isr_out.MatchFormatToFilename(argv[2]);
  isr_out.open(argv[2]);
  if (!isr_out) {
    cerr << "Unable to open file " <<  argv[2] << endl;
    exit(-1);
  }

  isr_out << *data;
}
```

Figure 9: ISRconvert uses the MatchFormatToFilename method of token streams to set token stream formats to match the filename extensions provided by the user. As a result, this simple program that reads in a file data and then writes it back out again can be used to convert a file from ascii to binary or vice-versa. It can even translate files between foreign file formats, or convert foreign formats to/from ISR3.1 formats.

```
void Line2D::output (Isrostream& dest)
{
  dest << "Line2D from (" << x1 << ", " << y1 << ") to (";
  dest << x2 << ", " << y2 << ")\n";
  dest << "Theta = " << theta << ", Contrast = " << contrast;
  dest << ", Disp = " << dispersion << "\n";
  dest << "Length = " << length << "\n";
}
```

Figure 10: Output function for the Line2D token class. If the Isrostream is an ascii stream, this will produce a human-readable file in which every feature value is preceeded by the feature name, and numbers are written in ascii. If the Isrostream is binary, on the other hand, none of the feature names or other character string will be printed, and all feature values will be written in binary.

whether the stream is BINARY or ASCII. If the stream is ASCII is prints the string, but if the stream is binary the string is ignored.

The reason for defining the Isrostream class is to make it easy to write a single output(Isrostream&) function that writes in both ascii and binary formats. For example, Figure 10 shows the output function for the Line2D token class discussed earlier. If the Isrostream argument *dest* is an ascii stream, it will print not only the endpoints of the line and other feature values, but it will preface each value with the string giving the feature name; the numbers, of course, will be written in ascii. The result is a file that is easy for people to read, even though it is very large. If *dest* is a binary stream, on the other hand, none of the feature names or other character strings (including newlines) will be printed, and the numbers themselves will be printed in binary. The result is a file that is usually about one fifth the size of the corresponding ascii file.

One disadvantage of Isrostreams occurs when a token has a character string field, the value of which should be written even to a binary file. In this case the output function should use the **print_string(char *)** method of the Isrostream (or **write(char *, int)**) to print the character stream regardless of format.

```
void Line2D::input (Isristream& source)
{
  source >> "Line2D from (" >> x1 >> ", " >> y1 >> ") to (";
  source >> x2 >> ", " >> y2 >> ")";
  source >> "Theta = " >> theta >> ", Contrast = " >> contrast;
  source >> ", Disp = " >> dispersion;
  source >> "Length = " >> length;
}
```

Figure 11: The input function for the Line2D class. The only difference to the body of the code between this and **Line2D::Output(Isrostream&)** is the direction of the double arrows.

The **Input(Isristream&)** method is the inverse of output(Isrostream&); it reads tokens that have been written using output. Not surprisingly, the Isristream class is to a standard C++ istream what Isrostreams are to ostreams – that is to say they are similar, but have two formats, ascii and binary. In ascii mode they expect to read numbers written as strings of ascii digits, while in binary mode they expect binary numbers. Furthermore, Isristreams handle strings a special way: if *is* is an ascii Isristream, the line of code

```
is >> ''foo'';
```

expects to find the string "foo" (not including the quotation marks) at the current point in the file. If it does, it skips over the string, advancing the file pointer to the end of the string. If it doesn't find the string, it is an error, so that the file pointer is not advanced and the command *!is* will return true. Conversely, if the Isristream is in binary mode, the above line code is a no-op.

As a result, in many cases it is possible to define the input function just by "turning around" the operators in the output function, as in Figure 11, which shows the input function for the Line2D class. The only significant difference between the bodies of code in Figure 10 and Figure 11 is the direction of the double arrows: ">>" vs. "<<".

Since the Line2D class is a simple token structure that does not contain pointers to any other tokens, there is no reason to define a trace() function for

```
class Line2D : public Token {
 public:
  float x1;
  float y1;
  float x2;
  float y2;
  float theta;
  float contrast;
  float dispersion;
  float length;

  void output(Isrostream& os);
  void input(Isristream& is);
  char *name() { return "Line2D"; }
};
```

Figure 12: A definition of the Line2D class that supports file I/O as well as storage and retrieval.

it. We can therefore expand the Line2D token definition shown in Figure 1 to the one in Figure 12 that will support I/O operations over line segments.

To give an example of how trace functions are defined, lets consider the Line2DPair token class. Line2DPair tokens describe the relationship between a pair of line segments, in terms of their point of intersection (itself an ISR3.1 token whose definition is not given here), a non-symmetric transform between the two lines called the UV transform (again a token), and various measures of overlap, as described in [4]. As shown in Figure 13, the class definition for Line2DPair includes pointers to the line segments that make up the pair. As a result, a trace function must be defined for Line2DPairs so that when a pair is written to a file, the component line segments are also included in the file. The trace method definition for Line2DPairs is shown in Figure 14.

The **output(Isrostream&)** function for Line2DPairs is straightforward and requires no special provisions; the fields that are pointers to tokens are output using the $<<$ operator just like all the other fields. The **input(Isristream&)** function, however, does have to handle pointers to to-

```
class Line2DPair : public Token {
 public:
  Line2D *lineA;                    /*Line A of pair AB*/
  Line2D *lineB;                    /*Line B of pair AB*/
  UV_transform *UVtrans;            /*UV_transform of pair AB*/
  Line2DIntersection* intersection;/*Point of intersection*/
  float displacement;               /*Lateral displacement*/
  float displacementpix;            /*Lat disp in pixels*/
  float separation;                 /*Relative separation*/
  float separationpix;              /*Separation in pixels*/
  float delta_theta;                /*Rotation from A to B*/
  float TA_separation;              /*Separation along A
                                      from segment to inter.*/
  float TA_separationpix;           /*TA_Separation in pixels*/
  float TB_separation;              /*Separation along B
                                      from segment to inter.*/
  float TB_separationpix;           /*TB_Separation in pixels*/

  void trace();
  void input(Isristream&);
  void output(Isrostream&);
  char *name() { return "Line2DPair"; }
};
```

Figure 13: Class definition for a pair of image line segments (Line2Ds). Because the definition includes pointers to other tokens, a trace() function has to be defined.

```
void Line2DPair::trace ()
{
  Register(lineA);
  Register(lineB);
  Register(UVtrans);
  Register(intersection);
}
```

Figure 14: The trace() function definition for Line2DPair. The trace function tells ISR3.1 which tokens a Line2DPair points to.

kens as a special case, so that the values of the pointers are updated to reflect the new memory positions of the tokens in the file. Figure 15 shows the input function for Line2DPairs. Numeric token fields such as displacement and seperation are read using the >> operator, as in the Line2D case. Pointers to tokens, however, are read using the **read_token_ptr(Isristream&)** function, which returns the new memory address of the token being pointed to. (read_token_pointer is defined as returning **(Token *)**, so its result must be cast to a pointer to the appropriate type of token. If it is cast to the wrong type of token, unpredictable errors will result.) ISR3.1 guarentees that the structure of a data tree is not disturbed by writting it out and then reading it back in again, so that if tokens A and B both pointed to token C in the data tree that was writting out, A and B will point to a single token C when read back in, not to two seperate tokens with the same feature values.

5.3 Foreign File Formats

In addition to its native ascii and binary formats, ISR3.1 is able to read and write files in other data formats. These "foreign" formats are not general purpose, in that only certain types of tokens can be stored in them, but they are very useful for passing data to, or getting data from, other systems, such as KBVision or Khoros.

In general, foreign format interfaces are implemented using the read and write functions of the foreign system, and therefore the foreign system's code must be accessible. Viff files, for example, are red and written using the func-

```
void Line2DPair::input (Isristream& is)
{
  is >> "Line2DPair:: LineA";
  lineA = (Line2D *)read_token_ptr(is);
  is >> "LineB";
  lineB = (Line2D *)read_token_ptr(is);
  is >> "UVtrans";
  UVtrans = (UV_transform *)read_token_ptr(is);
  is >> "intersection";
  intersection =
    (Line2DIntersection *)read_token_ptr(is);
  is >> "Delta_theta" >> delta_theta;
  is >> "Displacement" >> displacement >> displacementpix;
  is >> "Seperation" >> separation >> separationpix;
  is >> "TA_seperation" >> TA_separation >> TA_separationpix;
  is >> "TB_seperation " >> TB_separation >> TB_separationpix;
}
```

Figure 15: The input function for Line2DPairs. Pointers to tokens are read using the **read_token_ptr(Isristream&)** function.

Extension	*Token Types*	*Host System*
.isra	all (ascii)	ISR3.1
.isrb	all (binary)	ISR3.1
.viff	BytePlane, IntPlane, FloatPlane	Khoros
.im	BytePlane, IntPlane, FloatPlane	KBVision
.tks	Line2D (sets)	KBVision

Table 1: File Formats currently recognized by ISR3.1.

tions provided with Khoros; users who do not have access to Khoros cannot read or write viff files. Similarly, users who do not have access to KBVision cannot read or write im or tks files. To users in the UMass Computer Vision group, however, this should not present a problem since both systems are available.

Table 1 shows the foreign file formats that are currently supported. For each format, it lists the file extension and the types of tokens that can be stored in it, as well as the system for which the format is native. New foreign file formats can be (and are being) added to ISR3.1, but a description of how to add them is beyond the scope of this document.

Since the MatchFormatToFilename method of token streams knows about all of the file extensions in Table 1, the ISRconvert function shown in Figure 9 can be used to translate between compatable foreign file formats as well. For example, it could be used to convert a Khoros viff file into a KBVision im file, or to convert either to isra or isrb. Moreover, as long as applications programmers use MatchFormatToFilename to set stream formats, applications should work equally well on any format of data, often eliminating the need to explicitly convert file types.

6 Graphics Methods

This section explains how to add graphics methods to new token classes. Readers who are only interested in using predefined classes should skip this section.

One of the most important facilities ISR3.1 provides users is graphics. ISR3.1 includes a stand-alone executable graphics program called **xisrdisplay** that reads and graphically displays the contents of data files. It allows users to examine data [10], overlay data, and interactively manipulate data with a mouse. By using ISR3.1's IPC mechanisms (currently under development), users can also create interactive displays of executing programs. By far the most important feature of **xisrdisplay**, however, is its extensibility. New token types can be easily integrated by defining either a *Draw()* method (for vector tokens) or a *BitDraw()* method (for rasterized tokens). This section describes how to define these methods for new token types, and gives examples. (Section **??** describes **xisrdisplay** itself and how to use it.)

6.1 Vector Tokens

For display purposes, tokens can be roughly divided into two categories: *vector* tokens that are composed of geometric primitives such as lines and circles and can be displayed by X11 draw commands, and *raster* tokens such as images that are displayed in terms of bitmaps. For vector tokens class designers should provide a *Draw()* method, while for raster tokens a *BitDraw()* method should be defined. (BitDraw() takes more arguments that Draw() because displaying bitmaps requires extra information about the size and depth of the window.) Note that class designers should not define both Draw() and BitDraw(), as that will cause tokens to be displayed twice.

Draw() is a virtual function that takes eight arguments and returns void (as shown in Figure 16), but draws the token to the screen as a side-effect. The first three arguments are the X11 structures required by Xlib drawing primitives such as XDrawPoint, XDrawLine and XDrawArc. Since xisrdisplay allows users to interactively set the window size, zoom factor and graphics context there is no reason for the Draw() method to alter any of these arguments. The fourth argument is the (2D-to-2D) transformation that maps the token coordinate system onto the window coordinate system, while the last four arguments are the coordinates of the part of the window that is being redrawn in token (rather than window) coordinates.

The critical arguments are the token-to-window transformation and the boundary of the window portion being (re)drawn. In general, the coordinate system of a token is specified by the CoordSys object associated with the TokenSet or image it is a part of; this may or may not match the coordinate

[10] **xisrdisplay** is intended for 2D data only. A utility for displaying 3D data will hopefully be developed in the future.

```
/*----------------------------------------
  display  -- X11 display object pointer
  drawable -- X11 drawable (opaque pointer)
  gc       -- X11 graphics context
  trans    -- Transform2D object
  xmin     -- minimum X coord to display
  ymin     -- minimum Y coord to display
  xmax     -- maximum X coord to display
  ymax     -- maximum Y coord to display
  ----------------------------------------*/

void Token::Draw(Display *display, Drawable drawable, GC *gc,
                 Transform2D *trans, double xmin, double ymin,
                 double xmax, double ymax);
```

Figure 16: The Draw method for vector graphics. The first three arguments are passed directly to Xlib functions, the fourth defines the token to window coordinate transformation, and the last three specify the area being (re)drawn.

system of the window (which X11 defines as having (0,0) in the upper left corner). Moreover, the mapping from token coordinates to window coordinates changes interactively as the user zooms and roams around the image. The fourth argument to Draw() is a pointer to a Transform2D object that specifies how the token coordinates should be mapped onto the display window. Consequently, the apply() method of Transform2D will convert token (x, y) coordinates into window coordinates that can be used in the Xlib draw commands, as shown in Figure 17.

The last four arguments to the Draw method specify the rectangular region of the window that is being (re)drawn. These should be used for clipping purposes, since it is wasteful to draw tokens that are outside the boundary of the window. The rectangle is specified in token (rather than window) coordinates to avoid the cost of transforming token coordinates if they are outside the range being drawn. The Draw method shown in Figure 17 shows a particularly simple form of clipping, but it will still try to draw some tokens that are not visible. Better clipping algorithms can be found in any graphics book.

6.2 Raster Tokens

Some tokens, such as images, should be drawn not by making repeated calls to Xlib draw routines, but by creating a bitmap and displaying that. Such tokens are called raster tokens, and instead of defining a Draw() method, the class designer specifies a BitDraw() method. The idea behind BitDraw is the same as the idea behind Draw: it a the function by which **xisrdisplay** draws a token. Drawing a bitmap requires additional information about the depth of the display window (in bits) and the size of window being drawn to. BitDraw therefore has additional arguments not required by Draw, as shown in Figure 18.

Internally, BitDraw() methods work by creating an X bitmap and drawing it via the XPutImage() Xlib function. Generating and manipulating bitmaps tends to be complex, and no example is given here; readers are referred to their favorite X11 manual instead. Suffice it to say that only readers who are comfortable with X11 should attempt to define a BitDraw method.

```
void Line2D::Draw(Display* display, Drawable drawable, GC gc,
                  Transform2D* trans, float xmin, float ymin,
                  float xmax, float ymax)
{
  double *pts1, *pts2;

  /* clipping */
  if ((x1 < xmin) && (x2 < xmin)) return;
  if ((x1 > xmax) && (x2 > xmax)) return;
  if ((y1 < ymin) && (y2 < ymin)) return;
  if ((y1 > ymax) && (y2 > ymax)) return;

  /* calculate window coordinates */
  pts1 = trans->Apply(x1, y1);
  pts2 = trans->Apply(x2, y2);

  /* draw line */
  XDrawLine(display, drawable, gc,
            (int)rint(pts1[0]), (int)rint(pts1[1]),
            (int)rint(pts2[0]), (int)rint(pts2[1]));

  /* free window coordinate space */
  free(pts1);
  free(pts2);
}
```

Figure 17: The Draw method of Line2D. Applying the Transform2D converts token coordinates to window coordinates, while the last four arguments specify the rectangle that is being (re)drawn in token coordinates.

```
/*----------------------------------------
   display  -- X11 display object pointer
   drawable -- X11 drawable (opaque pointer)
   gc       -- X11 graphics context
   trans    -- Transform2D object
   ulx      -- upper left X coord to display
   uly      -- upper left Y coord to display
   lrx      -- lower right X coord to display
   lry      -- lower right Y coord to display
   visual   -- X11 visual of the screen drawable is on
   depth    -- depth of the display (number of display planes)
   cmap     -- colormap for the current window
----------------------------------------------*/

void Token::BitDraw(Display *display, Drawable drawable,
                    GC *gc, Transform2D *trans,
                    double ulx, double uly,
                    double lrx, double lry,
                    Visual *visual, int depth,
                    Colormap cmap);
```

Figure 18: The BitDraw method for vector tokens. The first four arguments are the same as for Draw(); the remaining seven are for compatibility with XPutImage().

6.3 Non-graphical Tokens

Although the preceeding discussion has divided tokens into those with vector graphics and those with raster graphics, some tokens types have no graphical properties at all. The Transform2D token class, for example, represents transformations from one 2D coordinate system to another. Transformations as are abstract entities that cannot be easily drawn, and consequently the Transform2D object has neither a Draw nor a BitDraw method defined. Since the default definitions of Draw and BitDraw (inherited from Token) are no-ops, drawing a Transform2D object produces no graphics at all.

Another approach is to draw an object by drawing its components. The Draw and BitDraw methods of the TokenSet class, for example, simply call the Draw and BitDraw methods for every token in the token in the TokenSet. (Note that this is the only circumstance under which it makes sense to define both a Draw and a BitDraw method, since presumably every token in the TokenSet will have one or the othe method defined, but not both.) Thus it is not neccessary for every token to produce graphical output, although most tokens should.

7 Utilities

8 Object Interfaces

This section contains the public interfaces for the objects described earlier in the manual, for reference purposes.

8.1 Isrstreams

Note: Isrstreams are the objects by which ISR3.1 implements its native file formats. They should not appear inside application programs for I/O; use TokenStreams for that. See Section 5.2 for a description of Isrstreams, and Section 5.1 for a description of TokenStreams.

```
class Isrstream {
 public:
  int format;         /* BINARY or ASCII */
```

```
  char *name;          /* filename associated with stream */
};

class Isrostream : public Isrstream {
  friend Isrostream& operator << (Isrostream&, const short);
  friend Isrostream& operator << (Isrostream&, const int);
  friend Isrostream& operator << (Isrostream&, const long);
  friend Isrostream& operator << (Isrostream&, const float);
  friend Isrostream& operator << (Isrostream&, const double);
  friend Isrostream& operator << (Isrostream&, const char*);
  friend Isrostream& operator << (Isrostream&, const char);
  friend int operator! (Isrostream&);

 public:
  Isrostream(char *filename, int stream_format = BINARY,
             int stream_type = ISR);
  Isrostream(int stream_format = BINARY,
             int stream_type = ISR);;
  ~Isrostream();

  void print_string(char *);
  Isrostream& write(const char *s, int n);
};
```

Isrostream& **operator** <<(Isrostream& os, const short val)

Isrostream& **operator** <<(Isrostream& os, const int val)

Isrostream& **operator** <<(Isrostream& os, const long val)

Isrostream& **operator** <<(Isrostream& os, const float val)

Isrostream& **operator** <<(Isrostream& os, const double val)

Print numbers in their ASCII representation if *os* is an ascii stream, or in their binary representations if *os* is a binary stream.

Isrostream& **operator** <<(Isrostream& os, const char *string)

Print *string* if *os* is an ascii string, otherwise do nothing.

Isrostream& **operator** <<(Isrostream& os, const char ch)

Print *ch*, no matter whether *os* is ascii or binary.

Isrostream& **operator !**(Isrostream& os)

Returns 1 if *os* is in an error state, 0 otherwise. See your favorite C++ book for a discussion of error states for C++ streams.

void **Isrostream::print_string**(char *string)

Print *string* to **this*, whether the format of **this* is ascii or binary. (This is used to write char * token fields to files, since the << operator will ignore strings when *os* is binary.)

void **Isrostream::write**(const char *string, int n)

Print the first *n* characters of *string* to **this*, reagrdless of the format of **this*.

```
class Isristream : public Isrstream {
  friend class ISRIStream;
  friend Isristream& operator >> (Isristream&, short&);
  friend Isristream& operator >> (Isristream&, unsigned short&);
  friend Isristream& operator >> (Isristream&, int&);
  friend Isristream& operator >> (Isristream&, unsigned int&);
  friend Isristream& operator >> (Isristream&, long&);
  friend Isristream& operator >> (Isristream&, unsigned long&);
  friend Isristream& operator >> (Isristream&, float&);
  friend Isristream& operator >> (Isristream&, double&);
  friend Isristream& operator >> (Isristream&, const char*);
  friend Isristream& operator >> (Isristream&,  onst char);
  friend int operator! (Isristream&);

 public:
  TypeDirectory *tokentype_table;
```

```
  char *read_string(int);
  Isristream& read(char *ptr, int n);
  void putback(char);
  char get();

  Isristream(char *filename, int stream_format = BINARY,
             int stream_type = ISR);
  Isristream(int stream_format = BINARY,
              int stream_type = ISR);
  ~Isristream();
};
```

Isristream& **operator** >>(Isristream& is, short val)

Isristream& **operator** >>(Isristream& is, unsigned short val)

Isristream& **operator** >>(Isristream& is, int val)

Isristream& **operator** >>(Isristream& is, unsigned int val)

Isristream& **operator** >>(Isristream& is, long val)

Isristream& **operator** >>(Isristream& is, unsigned long val)

Isristream& **operator** >>(Isristream& is, float val)

Isristream& **operator** >>(Isristream& is, double val)

Read numbers from files according to the format of *is*, ascii or binary.

Isristream& **operator** >>(Isristream& is, const char *string)

If *is* is in ascii format, skip over the string *string* in the input file, and position the file pointer at the end of the string. If the contents of the file do not match string, signal an error. If *is* is in binary format, do nothing.

Isristream& **operator** >>(Isristream& is, const char ch)

Read the next character from *is* and compare it to *ch*. If they are not equal, unget the read character and put the stream into an error state. (Note that the format of the stream is not relevent.)

char **Isristream::get**()

Read one character from **this*.

void **Isristream::putback**(char ch)

Put one character back into the input buffer of **this*.

Isristream & **Isristream::read**(char *ptr, int n)

ptr should be a pointer to a string of at least *n* characters. This function reads the next string (up to *n* characters from **this*, storing them in ptr.

char * **Isristream::read_string**(int n)

Allocates a string of length *n*, and then reads the next string from **this* into it. If it is unable to allocate memory or the string in the file is longer than *n*, read_string will return NULL, otherwise it returns a pointer to the buffer it filled.

8.2 Tokens

Tokens are the top-level object from which all other classes of ISR3.1 tokens are derived. The class Token mostly serves as a virtual function interface that defines the capabilities a visual token should have. See Section 2 for a more detailed description.

```
class Token {
  friend class ISROStream;
  friend class ISRIStream;
  friend Isrostream& operator<< ( Isrostream&, Token&);
  friend Isrostream& operator<< ( Isrostream&, Token*);
  friend Isristream& operator>> ( Isristream&, Token&);
```

```
 public:
  Token();
  virtual ~Token();

  /* Virtual functions that MUST be defined.*/
  virtual char* name();

  /* Returns the integer type of any token */
  int type_index ();

  /* required for I/O capability */
  virtual void output(Isrostream&);
  virtual void input(Isristream&);
  virtual void trace() { return; }
  /* required for graphics capability */
  /* NOTE: either Draw or BitDraw should be defined,
     but not both. */
  virtual void Transform(Transform2D *trans);
  virtual void Draw(Display* display, Drawable drawable, GC gc,
                    Transform2D* trans, float xmin, float ymin,
                    float xmax, float ymax);
  virtual void BitDraw(Display* display, Drawable drawable,
                       GC gc, Transform2D* trans,
                       float xmin, float ymin,
                       float xmax, float ymax,
                       Visual *visual, unsigned int depth,
                       Colormap cmap);
  /* required for TKS compatibility & ISRAccess/ISRPut */
  virtual int field_count() { return 0; }
  virtual TokenField *fields() { return NULL; }
};
```

virtual char * **Token::name**()

Every token must have a name function that identifies its class, for purposes of retrieval (accessing only tokens of the correct type) and I/O (identifying the type of object to be created when a pointer is encountered). The only requirement of a class name is that it be a character string that is different from all other class names in the system.

int **Token::type_index**()

Unlike most of the functions defined for Token, type_index() is not a virtual function that the designer of a subclass must implement – it is already implemented. Type_index() returns the integer associated with the class type of a token. Within an executable module, indexes are consistent, so that two instances will have the same type index if and only if they are instances of the same class. These indexes are not necessarily consistent across different executable modules, however, and they may change anytime the system is recompiled. Therefore it is not recommended for a user to write these indices to a file or do anything with them other than comparitive tests between tokens; in general, name() should be used to check the identity of a token class.

virtual void **Token::input**(Isristream& is)

virtual void **Token::output**(Isrostream& os)

These virtual functions define how instances of a token class are read from and written to files. It is important that the two methods be inverses of each other, so that input() can read what output() writes, in both ascii and binary modes. Note that input() and output() are only invoked for ISR3.1-format files (.isra or .isrb), not when reading or writing foriegn file formats.

In general, it is a good idea for output() to produce output that can be easily read by a human when the Isrostream is in ascii mode, and that is more compact when in binary mode. See Section 5.2 for a description and example of how this can be accomplished.

virtual void **Token::trace**()

This function tells the system which tokens another token points to; this is particularly important during I/O, when the system needs to be able to trace through a data tree in order to print it.

If a token does not contain pointers to any other token, then its trace() function does not need to be redefined (the default operation is a no-op).

Otherwise, its trace function should call the function **Register(Token *tok)** for every token it points to. (If tok is a NULL pointer, Register(tok) is a no-op.) Figure 14 gives an example.

> virtual void **Token::Draw**(Display* display, Drawable drawable, GC gc, Transform2D* trans, float xmin, float ymin, float xmax, float ymax)

Draws the token *this* to *display*, using drawable *drawable* and graphics context *gc*. (Readers who are not familiar with X11 should not worry about the intracacies of display, drawables and contexts, other than knowing that these are the first three arguments of Xlib draw routines such as XDrawLine, XDrawPoint, XDrawArc and XDrawText.) The *trans* argument specifies the token to window coordinate transformation; An (x, y) coordinate in token space can be converted to window coordinates by using the Apply() method of Transform2D's (see the example in Figure 17). The last four arguments specify the coordinates (in token space) of the rectangle being redrawn. These are useful four clipping purposes, since it is more efficient not to draw tokens that lie outside of this rectangle.

> virtual void **Token::BitDraw**(Display* display, Drawable drawable, GC gc, Transform2D* trans, float ulx, float uly, float lrx, float lry, Visual *visual, unsigned int depth, Colormap cmap)

The BitDraw method renders a token into an X11 bitmap, and then displays that bitmap to the window *display*, usually using the XPutImage() Xlib routine. Three first four arguments are the same as for Draw(); the remaining seven are in the form expected by XPutImage().

8.3 TokenStreams

TokenStreams are top-level objects for token I/O in ISR3.1. They define streams, similar to ostreams and istreams in standard C++, to which tokens can be written and read, in any of the various data formats supported by ISR3.1. See Section 5.1 for a description of TokenStreams and their use.

```
class TokenStream {
 public:
  char *name;                    /* filename or stream name */
```

```
  int openp();
  int MatchFormatToFilename(char *filename);
  virtual int SetFormat(StreamFormat);
};
```

int **TokenStream::openp**()

Returns 1 if the stream **this* is open, 0 otherwise.

int **TokenStream::MatchFormatToFilename**(char *filename)

Sets the format of a stream to match the extension of a filename, for example when the filename ends in .isra, the stream is set to be an ISR3.1 ascii stream. The complete set of known file extensions is given in Table 1.

```
class TokenOStream : public TokenStream {
  friend TokenOStream& operator << (TokenOStream&, Token&);
  friend TokenOStream& operator << (TokenOStream&, Token*);
  friend int operator ! (TokenOStream&);

 public:
  int SetFormat(StreamFormat);

  void open(char *);
  void close();

  TokenOStream();
  TokenOStream(StreamFormat);
  TokenOStream(char *);
  virtual ~TokenOStream();
};
```

TokenOStream& **operator** <<(TokenOStream& os, Token& tok)

TokenOStream& **operator** <<(TokenOStream& os, Token* tok)

Writes a token to a TokenOStream. Ideally, the stream should have been opened previously; however, if a filename has been specified for *os*, the << operator will open the stream (using **TokenOStream::open**) before writing.

Tok can be either a token or a pointer to a token, it makes no difference. << returns a TokenOStream reference so that instances of it can be chained together, as in

```
os << token1 << *token2;
```

(assuming token1 and token2 are pointers to tokens).

int **operator !**(TokenOStream& os)

Returns 1 if token stream *os* is in an error state, 0 otherwise. Error states can be created by a variety of conditions, such as exceeding one's disk quota, etc. Refer to your favorite C++ manual for a discussion of error states and streams.

void **TokenOStream::close**()

Closes stream **this*. It is an error to close a stream that was not previously open. Once closed, a stream may be reopened with a new filename.

void **TokenOStream::open**(char *filename)

Open the file for writing. If the format of **this* has not yet been specified, use MatchFormatToFilename to set the stream format to match *filename.*

The exact behavior of this function depends on the format of the stream, but it will always generate an error if the file cannot be accessed for writing.

int **TokenOStream::SetFormat**(StreamFormat frmt)

Sets the format of a TokenOStream to one of the types given in Table 1. The argument is a StreamFormat, which is an enumerated data type of the extensions (minus the periods) found in Table 1, such as **isra, isrb, viff** and **im**.

```
class TokenIStream : public TokenStream {
  friend TokenIStream& operator >> (TokenIStream&, Token&);
  friend int operator ! (TokenIStream&);

 public:
  int SetFormat(StreamFormat);

  void open(char *);
  void close();
  Token* read_token_ptr();

  TokenIStream();
  TokenIStream(StreamFormat);
  TokenIStream(char *);
  virtual ~TokenIStream();
};
```

TokenIStream& **operator** >>(TokenIStream& is, Token& tok)

Fills *tok* with data read from TokenIStream *is*. If the file contains the wrong type of token, *tok* will not be modified and *is* will be put into an error state. (Check for error states using the ! operator.) Ideally, the stream should have been opened previously; however, if a filename has been specified for **is*, the >> operator will open the stream (using **TokenIStream::open**) before reading.

>> returns a TokenIStream reference so that instances of it can be chained together, as in

```
os >> token1 >> token2;
```

(assuming token1 and token2 are tokens).

int **operator !**(TokenIStream& is)

Returns 1 if token stream *is* is in an error state, 0 otherwise. Error states can be created by a variety of conditions, including trying to read one type of token into another. Refer to your favorite C++ manual for a discussion of error states and streams.

void **TokenIStream::close**()

Closes stream **this*. It is an error to close a stream that was not previously open. Once closed, a stream may be reopened with a new filename.

void **TokenIStream::open**(char *filename)

Open the file for reading. If the format of **this* has not yet been specified, use MatchFormatToFilename to set the stream format to match *filename*.

The exact behavior of this function depends on the format of the stream, but it will always generate an error if the file cannot be accessed for reading.

Token * **TokenIStream::read_token_ptr**()

Read a token (of arbitrary type) from the TokenIStream and return a pointer to it. Ideally, the stream should have been opened previously; however, if a filename has been specified for **is*, read_token_ptr() will open the stream (using **TokenIStream::open**) before reading.

int **TokenIStream::SetFormat**(StreamFormat frmt)

Sets the format of a TokenIStream to one of the types given in Table 1. The argument is a StreamFormat, which is an enumerated data type of the extensions (minus the periods) found in Table 1, such as **isra, isrb, viff** and **im**.

8.4 TokenSets

The public definitions of the *TokenSet* class structure is as follows

```
class TokenSet : public Token {
 public:
  CoordSys2D coords;

  void output(Isrostream& os) {coords.output(os); container_output(os);}
  void input(Isristream& is) {coords.input(is); container_input(is);}
  void trace();
  void Transform(Transform2D *trans);
  void Draw(Display* display, Drawable drawable, GC gc,
          Transform2D* trans,
```

```
            float xmin, float ymin, float xmax, float ymax);

// Set operations:
  TokenSet& Union( TokenSet * );
  TokenSet& Intersect( TokenSet * );
  TokenSet& Diff( TokenSet * );

  TokenSet* select( int );
  TokenSet* select( char * );
  TokenSet* retrieve( int , char * );
  TokenSet* retrieve( int, int , char * );
  TokenSet* retrieve( int, int );
  TokenSet* retrieve( char *, int, int );
  TokenSet* retrieve( int, int, int );
  TokenSet* retrieve( char *, int, int, int );
  TokenSet* retrieve( int, float, float );
  TokenSet* retrieve( int, int, float, float );

  virtual int count();
  virtual int insert(Token *);
  virtual int add(Token *);
  virtual int remove(Token *);
  virtual int member(Token *);

  virtual TokenSet *new_inst();
  virtual TokenSetState* state();
  virtual TokenSetState* state(Token *);
};
```

The implementation of the functions defined for the **TokenSet** class is different for the two classes at the next level of the hierarchy, but the functionality remains the same. Therefore, the description of these functions is given at this level and the user who understands the concepts of inheritance and is not interested in implementation details can acquire a perspective of the hierarchy.
Note: An object of type TokenSet cannot be created legally, since there are pure virtual functions in the definition of the TokenSet class. *TokenArray*s or *TokenList*s are the only objects of type—or inheriting from—TokenSet which can exist. Therefore, some of the examples are delayed until we introduce the two classes which one can create an instance of, while some examples are given in which objects of type TokenSet are used, assuming that the user is aware of this fact. The user is also expected to be familiar with C++.

8.4.1 Selection and Retrieval Functions

TokenSet* **TokenSet::select**(int type)

select returns a TokenSet of tokens in **this* of type *type*.

TokenSet* **TokenSet::select**(char *name)

select returns a TokenSet of tokens in **this* having name *name*.

TokenSet* **TokenSet::retrieve**($\cdots$)

Several *retrieve* functions exist, which return a subset of a TokenSet consisting of the tokens satisfying certain conditions. The condition is that one of the fields (the one *offset* bytes from the beginning of the structure defining the token) have a certain value. This *offset* and *value* are passed as arguments. A different function has to be used depending on the type of this *value*. One can restrict the search for the tokens that satisfy the condition to a single type of token. This *type* has to be specified by passing the type—or for some functions the name—of the token to the *retrieve* function. The following is a list of available retrieve functions:

TokenSet* **TokenSet::retrieve**(int offset, char *)

TokenSet* **TokenSet::retrieve**(int type, int offset, char *)

TokenSet* **TokenSet::retrieve**(int offset, int)

TokenSet* **TokenSet::retrieve**(char *name, int offset, int)

TokenSet* **TokenSet::retrieve**(int type, int, int)

TokenSet* **TokenSet::retrieve**(char *name, int offset, int min, int max)

TokenSet* **TokenSet::retrieve**(int offset, float, float)

TokenSet* **TokenSet::retrieve**(int type , int offset, float min, float max)

The last argument(s) in each case is (are) the *value* argument(s). The ones with two int or float arguments return all the tokens of which the value of the comparison field (specified by offset) falls in the range [min, max].

8.4.2 Set Functions

TokenSet& **TokenSet::Union**(TokenSet *ct2)

Union stores the union of **this* and *ct2* in **this*. Two tokens are considered to be equal if the pointers pointing to them are equal, so this is not a content- or property-based union operation.
Similarly,

TokenSet& **TokenSet::Intersect**(TokenSet *ct2)

and

TokenSet& **TokenSet::Diff**(TokenSet *ct2)

store the intersection and difference of **this* and *ct2* in **this*.

8.4.3 Maintenance Functions

The virtual functions *count*, *insert*, *add*, *remove* and *member* are used to maintain a TokenSet. These functions are redefined at the next level of the hierarchy.

virtual int **TokenSet::count**()

returns the number of tokens stored in the TokenSet.

virtual int **TokenSet::insert**(Token *tk)

inserts *tk* into the TokenSet. *insert* does not check to see if *tk* is already there.

virtual int **TokenSet::add**(Token *tk)

inserts *tk* into the TokenSet if it is not already there.

virtual int **TokenSet::remove**(Token *tk)

removes *tk* from the TokenSet.

virtual int **TokenSet::member**(Token *tk)

returns 1 if *tk* is stored in the TokenSet.

8.4.4 TokenArray

An array-based mechanism is used to implement TokenSet storage and access functions in *TokenArrays*. Tokens can be accessed sequentially—as given in Section 3.1 or by directly indexing into the array:

```
TokenArray *ta;
Token *tk;
int i;

tk = ca[i];
```

The following is the *TokenArray* class definition:

```
class TokenArray : public TokenSet {
 public:
  TokenArray(int init_size = DEFAULT_ARRAY_SIZE,
                             int incr = DEFAULT_ARRAY_INCREMENT);
  virtual ~TokenArray();

  char *name() {return "TokenArray";}

  TokenSetState* state();
  TokenSetState* state(Token* token);

  int count() {return token_count;}
  int insert(Token *);
  int add(Token *);
  int remove(Token *);
  int member(Token *);
  TokenSet *new_inst() { return( (TokenSet *) new TokenArray ); }

  int index(Token *);
  Token* value (int index);
  Token* operator[] (int index);
  int array_size() {return size;}
  int max_index() { return max_free_index;}
  int extend_array();
  int extend_array(int new_size);
```

```
  TokenList *Array2List();
  TokenArray& sort();
  TokenArray& sort( int (*cmp)(Token *, Token *) );
};
```

char **TokenArray::TokenArray**(int init_size, init incr)

TokenArray returns an object of type TokenArray, initially of size *init_size*. *incr* is the size by which the TokenArray will grow everytime it is extended (see *extend_array* below).

char **TokenArray:: TokenArray**()

TokenArray frees the space allocated for a TokenArray object when the C++ function `delete` is called to return the storage allocated by a call to `new`.

char **TokenArray::name**()

name returns the name of this token, that is, "TokenArray".

The virtual maintenance functions *count*, *insert*, *add*, *remove* and *member* are instantiated here. These functions operate on the array structure. The usage and operation of these functions are the same as in the description given in Section 8.4.

TokenSet * **TokenArray::new_inst**()

new_inst returns a pointer to an object of TokenArray, cast into a pointer to TokenSet.

new_inst provides a general mechanism to create new objects of the same type as the object in hand. Functions that expect an object of any type that derived from TokenSet, but have to return an object of type TokenSet, can create new objects of the same type as the object passed to the function. This becomes important when it is desirable to keep the implementation of an object. For example, retrieval functions expect an object of type TokenSet. An object of type TokenArray or TokenList will be passed to such a function and will be automatically cast into TokenSet. The retrieve function no longer knows what type of an object was passed as an argument. **new_inst** is used to create an object of the same type as the object passed as an argument, cast into to TokenSet. An object of type TokenSet is returned, with the underlying structure preserved. The user has to cast back the result of the function into an object of desired type. This way, retrieval functions have uniform return types while the underlying structure (array or linked list) is kept in the returned object (see example in Figure 5).

int **TokenArray::index**(Token *tk)

index returns the index of *tk* in the TokenArray, that is `i;` such that `ca[i] == token`, where `ca` is a `TokenArray`.

Token * **TokenArray:value**(int index)

value returns the token `tk` such that `tk == ca[index]`.

Token * **TokenArray::operator**[](int index)

This function is equivalent to the **value** function, but makes it possible to treat TokenArrays as arrays, hiding the internal details.

int **TokenArray::array_size**()

array_size returns the current size of the array used to store tokens. *array_size* and *token_count* are the same only if the array is full.

int **TokenArray::max_index**()

In a TokenArray, there is usually more than one position where a new token can be inserted. There is an index after which the array is empty, and tokens can be stored starting from that position. That first index after which the rest of the array is uniformly empty is denoted **max_free_index**. This is the minimum index after which we are sure no token is stored in the array. *max_free_index* is useful because it limits the search in the array: we don't need to look past this index for tokens.

int **TokenArray::extend_array**([int new_size]) *extend_array* allocates some more space and extends the array to include this new space. If no size is specified, the array is extended by a fixed amount of 100 tokens, given by the constant DEFAULT_ARRAY_INCREMENT.

TokenList * **Container::Array2List**()

Array2List converts a TokenArray into a TokenList. The implementation of the storage mechanism is converted from an array to a linked list.

TokenArray **TokenArray::sort**([int (*cmp)(Token *, Token *)])

sort sorts the pointers to tokens in the TokenArray in ascending order.

8.4.5 TokenArrayState

```
class ArrayState : public TokenSetState {
 public:
  ArrayState(TokenArray* ta, int init_value = 0);
  Token* value() {return array_ptr->value(index);}
  Token* next();
};
```

The TokenArrayState functions **value** and **next** are instantiations of the funtions described in Section 3.1 and therefore identical in their usage and functionality to the *value* and *next* functions introduced for TokenSetStates.

8.4.6 TokenList

As in Section 8.4.4, the *TokenList* class instantiates the virtual Maintenance functions, with the functions operating on a linked-list structure this time.

The definition of the *TokenList* class is given as

```
class TokenList : public TokenSet {
 public:
  TokenList() { head = tail = NULL; token_count = 0;}
  virtual ~TokenList();

  char *name() {return "TokenList";}

  int count() {return token_count;}
```

```
  int insert(Token *);
  int add(Token *);
  int remove(Token *);
  int member(Token *);
  TokenSet *new_inst() { return( (TokenSet *) new TokenList ); }

  TokenSetState* state() {return new TokenListState(head);}
  TokenSetState* state(Token *token) {return new TokenListState(locate(token));}

  TokenArray *List2Array();
  TokenList& sort();
  TokenList& sort( int (*cmp)(Token *, Token *) );
};
```

char **TokenList::TokenList**()

TokenList returns an object of type TokenList, initially empty.

char **TokenList:: TokenList**()

TokenList frees the space allocated for a TokenList object when the C++ function `delete` is called to return the storage allocated by a call to `new`.

char * **TokenList::name**()

name returns the name of this token, that is "TokenList".

The virtual maintenance functions *count*, *insert*, *add*, *remove* and *member* are instantiated here. These functions operate on the array structure. The usage and operation of these functions are the same as in the description given in Section 8.4.

TokenSet * **TokenArray::new_inst**()

new_inst returns a pointer to an object of type TokenList, cast into a pointer to TokenSet. See Section 8.4.4 above for the description and the usage of this method.

TokenSetState * **TokenList::state**([Token *token])

TokenList::state returns a TokenSetState pointer that can be used for purposes described in Section 3.1.

TokenArray * **TokenList::List2Array**()

List2Array converts a TokenList into a TokenArray. The implementation of the storage mechanism is converted from a linked-list to an array.

TokenList **TokenList::sort**([int (*cmp)(Token *, Token *)])

sort sorts the pointers to tokens in the TokenList in ascending order.

8.4.7 TokenListState

```
class TokenListState : public TokenSetState {
 public:
  TokenListState(TokenListElement *init = NULL) {tle = init;}
  Token* value() {if (tle != NULL) return tle->token; else return NULL;}
  Token* next() {if (tle != NULL) tle = tle->ptr; return value();}
};
```

The TokenListState functions **value** and **next** are instantiations of the funtions described in Section 3.1 and therefore identical in their usage and functionality to the *value* and *next* functions introduced for TokenSetStates.

8.5 Grids

The public definition of the *Grid* class structure is as follows

```
class Grid: public TokenSet {
public:
  Grid( int tokentype = -1,
        int xsize = DEFAULT_XSIZE, int ysize = DEFAULT_YSIZE,
        float x_init = DEFAULT_GRID_XINIT, float y_init = DEFAULT_GRID_YINIT,
        float x_inc = DEFAULT_GRID_XINC, float y_inc = DEFAULT_GRID_YINC,
        int pagesize = DEFAULT_PAGE_SIZE, CoordSys2D *coord_sys = NULL );
  ~Grid() {free( grid_cell );}

/* Methods to help different tools such as I/O, Draw, Transform: */
  char *name() {return "Grid";}
```

```
  TokenSetState* state();
  TokenSetState* state( Token *token );

/* Grid maintenance methods: */
  int setCellState(int x_coord, int y_coord, int state);
  void deactivate();
  int activateCell( int x_coord, int y_coord );
  int deactivateCell( int x_coord, int y_coord );

  int count() { return token_count; }
  int insert( Token *token );
  int add( Token *token );
  int remove( Token *token );
  int member( Token *token );

  TokenArray *retrieve();

/* Various tools provided: */
  int traceLine( float x1, float y1, float x2, float y2, int state);
  int gridifyLines( TokenSet *ts);

  int rasterizeRectangle( float x1, float y1, float x2, float y2 );
  int rasterizeCircle( float x0, float y0, float radius );
  int rasterizePolygon( COORDINATE points_list[MAX_POINTS],
                        int num_of_points );
};
```

8.5.1 Basic Functions

Grid::Grid(int tokentype, int xsize, int ysize, float x_init, float y_init, float x_inc, float y_inc, int pagesize, CoordSys2D *coord_sys)

Grid returns a 2 dimensional grid. *xsize* and *ysize* specify the number of cells in the grid in each dimension, while *x_inc* and *y_inc* specify the size of a grid cell in each dimension. *x_init* and *y_init* indicate the starting point (lowest coordinates) of the grid. *pagesize* is the size of the internal memory pages used to store tokens in a grid cell. (Since a cell can use multiple pages, *pagesize* is not a bound on the number of tokens per cell, although it should be larger than the expected number of tokens per cell to be efficient. Users who are not concerned with efficiency can use the constant GRIDPAGESIZE.) The final argument, *coord_sys*, is a pointer to a CoordSys2D structure which specifies the coordinate system properties.

char* **Grid::name**()

name returns the name of this token, that is, “Grid”.

TokenSetState* **Grid::state**()

TokenSetState* **Grid::state**(Token *token)

As with *TokenSets*, the *state* function returns a one-directional index into a grid. The elements in a grid can be visited one by one with the help of the **TokenSetState::next** and **TokenSetState::value** functions. The optional argument *token* specifies a token to which the state index will point initially; the visit to tokens will start from this specified token. See Section 3.1 for a more detailed description.

8.5.2 Maintenance Functions

int **Grid::setCellState**(int x_coord, int y_coord, int state)

setCellState is a helper function for the activation and deactivation functions. The cell at position (int x_coord, int y_coord) is activated if *state* is ACTIVE and it is deactivated if *state* is INACTIVE.

void **Grid::deactivate**()

deactivate switches the states of all the CellElements in the grid to INACTIVE. None of the cells will be considered if *retrieve* is called immediately after a call to this method. (See activateCell() for a description of active vs. inactive cells.)

int **Grid::activateCell**(int x_coord, int y_coord)

activateCell activates cell (x_coord, y_coord) of grid. Once a cell is active, calls to **Grid::add** will cause tokens to be stored in the cell, and calls to **Grid::retrieve** will cause them to be retrieved from the cell. Many cells can be active at once, and a cell remains active until **Grid::deactivateCell** or **Grid::deactivate** is called. For **grid** declared as “Grid *grid;”, if

$0 \leq x\_coord \leq grid \rightarrow \text{x\_size}$ and $0 \leq y\_coord \leq grid \rightarrow \text{y\_size}$, *activateCell* returns 0, otherwise *activateCell* returns -1.

int **Grid::deactivateCell**(int x_coord, int y_coord)

deactivateCell deactivates the state of the indicated grid cell. Tokens in the cell will no longer be returned by **Grid::retrieve** (unless the cell is reactivated later), and **Grid::add** will no longer store a token to the cell. If $0 \leq x\_coord \leq grid$ →x_size and $0 \leq y\_coord \leq grid$ →>y_size, ISRdeactivateCell returns 0, otherwise it returns -1.

int **Grid::insert**(Token *token)

insert adds *token* to every ACTIVE cell of grid regardless of whether it is already stored in the cell. *insert* does not affect the status of grid cells, so that any cells that were ACTIVE before a call to *insert* are still ACTIVE after it.

int **Grid::add**(Token *token)

add is similar to *insert* but *token* is added to every ACTIVE cell of grid only if it is not already stored in the cell. The token can be of any type. *add* does not affect the status of grid cells, so that any cells that were ACTIVE before a call to *add* are still ACTIVE after it.

int **Grid::remove**(Token *token)

remove removes *token* from every ACTIVE cell in which it is stored. A list of cell indeces is kept for each token, which protects this function from exhaustive search when the cells that a token belongs to have to be accessed.

int **Grid::member**(Token *token)

member returns 1 if *token* is stored in the grid.

TokenArray* **Grid::retrieve**()

retrieve retrieves all tokens from every ACTIVE cell in grid, and returns them in a TokenArray. The TokenArray will not contain any duplicate entries, but it may be empty.

8.5.3 Rasterization Functions

The routines for activating and deactivating cells and for storing and retrieving functions provide the basic operations for cells. Starting with these functions, a programmer can build up the neccessary routines for storing and retrieving geometric shapes. ISR3.1 includes rasterization routines, however, for mapping the common shapes onto grids, making it easy for users to store and retrieve spatially. *Warning: These functions make assumptions about certain token types, and may not be valid if the token definitions are changed.*

int **Grid::traceLine**(float x1, float y1, float x2, float y2, int state)

traceLine activates every cell touched by the line segment extending from (x1, y1) to (x2, y2).

int **Grid::gridifyLines**(TokenSet *ts)

gridifyLines stores each line in *ts* into the cells that the line passes through. This is done by activating the cells that the line crosses with a call to *traceLine* and *add*ing the line into the activated cells. The coordinate system of grid is set to be the coordinate system of *ts*. This assumes that tokens represented in some fixed coordinate system are being and will be stored in the grid.

int **Grid::rasterizeRectangle**(float x1, float y1, float x2, float y2)

rasterizeRectangle activates every cell touched by the scan-line aligned rectangle with corners in (x1, y1) and (x2, y2).

int **Grid::rasterizeCircle**(float x0, float y0, float radius)

rasterizeCircle activates every cell located within a circle of radius *radius* centered at the point (x0, y0).

int **Grid::rasterizePolygon**(COORDINATE points_list[MAX_POINTS], int num_of_points)

rasterizePolygon activates every grid cell inside of or on the boundary of a convex polygon. The polygon is specified as an array of COORDINATEs, and the points must be ordered in a walk around the polygon.

References

[1] *KBVision^{TM} System User's Guide version 2.5*, Amherst, MA, 1990.

[2] J. Argiro. *KHOROS documentation*, 1991.

[3] J. Brolio, B. Draper, R. Beveridge, and A. Hanson. The ISR: an Intermediate Symbolic Representation for Computer Vision. *IEEE Computer*, 22(12):22–30, 1989.

[4] G. Reynolds and R. Beveridge. Searching for Geometric Structures in Images of Natural Scenes. In *IUW, Los Angeles, CA,*, pages 257–271, Feb. 1987.