ISR3.2 User's Guide & Programmer's Reference

Bruce A. Draper

June 26, 1995

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

ISR3 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 [1, 2]. ISR3 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. Many of these systems integration problems can be traced to 1) incompatible structures for symbolic image data or 2) incompatible file formats. ISR3 supports (computer vision) system integration by supplying a 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). Recognizing that many useful vision software modules have already been developed that do not use ISR3 classes or I/O routines, ISR3's I/O routines also support reading and writing files in many commonly-used formats, including *gif, tiff, viff* and *im*.

ISR3 is not a software panacea, however. It cannot turn poorly written code into efficient and portable software modules, nor can the ISR3 library anticipate every data structure that a computer vision researcher might need. Instead, ISR3 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 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 researchers won't have to. If ISR3 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 as

researchers develop personal ISR3 class libraries, but if researchers who design new classes make them generally available, we can all benefit from each other's labors.

1.2 Input/Output (I/O)

The core of ISR3 is a library of C++ classes defining symbolic computer vision data structures, such as points, lines and faces. These classes are called *token types*, 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 is the notion that new token types can be introduced by programmers as necessary. Although ISR3 comes with an initial library of token types, including tokens for images, points, regions and lines, there has been no attempt to ennumerate all possible data structures for computer vision research. Instead, ISR3 is designed to be an extensible system that will grow as users define new token types.

For file I/O, ISR3 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 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 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) whenever a person wants to inspect their contents. Token streams can also be set to various foreign file formats, such *gif* (used in Mosaic and on many PCs), *tiff* (used on Apple MacIntoshes), *viff* (used by Khoros [6]) and *im* and *tks* (used in KBVision [8]). More foreign file formats, including *jpeg*, the *IUE data exchange format* and AutoCad's *dfx* format are planned (see Section 2.2.2).

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 (process) to another. An interprocess communication (IPC) module is currently under development for ISR3 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.

1.3 Token Sets

The most important capability of ISR3 is the ability to store tokens in sets, and then retrieve individual tokens or subset of tokens from those sets. ISR3 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 also provided for iterating through tokensets, and sets of tokens can be retrieved by their feature values (i.e. associatively). Special types of token sets may provide additional retrieval mechanisms; Grids, for example, allow tokens to be 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, and to store a single token in multiple tokensets.

1.4 Graphics

In addition to the class library, ISR3 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 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 to show intermediate results of a program as it is running.

1.5 Organization of this Document

After the overview, this document is divided into three sections. The *User's Guide* introduces programmers to the ISR3 class library, showing them how to use the file I/O, token retrieval and graphics capabilities provided. Note that ISR3 is a programming tool not an end-product, so the users referred to are programmers, not unsophisticated customers. The *Programmer's Reference* contains all the detailed information needed for programmers to extend ISR3 by designing new token types. Novice users will therefore want to start by reading the user's guide. The programmer's reference is needed if and when a programmer decides to extend ISR3 by implementing a new token type (class). Readers interested in a higher-level motivation for ISR3 should see [2].

Chapter 2

User's Guide

2.1 Tokens

ISR3 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, I/O and graphics capabilities are provided as methods (functions) associated with the tokens, whereas storage and retrieval facilities are provided by special objects called **TokenSets**. Tokens can be stored in tokensets and later retrieved by their attribute values, or by iterating through the tokenset. Tokens stored in one special type of TokenSet (called a **Grid**) can also be retrieved by their spatial location.

In general, the most important thing to remember about tokens is that they are just instances of C++ classes. As a result, they are created using the C++ **new** operator and destroyed using the **delete** operator. To access a field of a token, you use the C++ direct access (.) and indirect access (->) operators, just as you would with any other class instance. As a result, there is no ISR3 analogue to the ISR2 function *isr2::value()*[2].

For example, the **Line2D** token type shown in Figure 2.1 is used for (two dimensional) image line segments. It is a straightforward C++ class definition, and Line2D tokens can be created, accessed and deleted like any other C++ class instance. For example, Figure 2.2 shows a simple program for reading endpoints typed in by a user and storing them in a Line2D token.

There are several function in Figure 2.1 whose purpose may not be obvious to you. The **name** method, for example, returns the string "Line2D". All ISR3 token types include a virtual function called name that returns a string identifying the type of the token. This allows a program to identify the type of an otherwise unknown token. There is also a method called **type_index()** of the base class Token which returns an integer which is unique to the token type, and can be used to determine whether two tokens are of the same type. (The integer associated with a specific token type may change any time the system is recompiled, however, so user's are advised to use the name method if it is necessary to identify the type of a token.)

Two other methods of Line2D, **Transform** and **Draw**, will be explained later. The **output**, **input**, **field_count**, and **fields** methods are used internally by ISR3 and are discussed in the Pro-

[1]Inheriting from Token is a necessary but not sufficient condition for being an ISR3 token. The Programmer's Reference describes the complete set of requirements for designing new token types.

[2]Because many readers of this manual (particularly at UMass) are familiar with ISR2 as described in [1], this manual contains many inline compatibility notes.

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

  Line2D();
  Line2D(float X1, float Y1, float X2, float Y2);
  Line2D(float X1, float Y1, float X2, float Y2, float Theta, float Contrast,
 float Dispersion, float Length, long int Region);

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

  int field_count() { return LINE2D_FIELD_COUNT; }
  TokenField *fields() { return field_definitions; }

  void Transform(Transform2D *trans);
  void Draw(Display* display, Drawable drawable, GC gc, Transform2D* trans,
    float xmin, float ymin, float xmax, float ymax);
};
```

Figure 2.1: The (public portion of the) Line2D token type definition

```
Line2D *make_Line2D()
{
   Line2D *ptr = new Line2D;

   cout << ''Please type in the endpoint coordinates:'';
   cin >> ptr->x1 >> ptr->y1 >> ptr->x2 >> ptr->y2;

   return ptr;
}
```

Figure 2.2: An example program using Line2D tokens. This program reads endpoints typed by a user and creates the corresponding Line2D token.

grammer's Reference. Note that the output and input methods should NOT be used by application programmer's for I/O.

2.2 File I/O

2.2.1 TokenStreams

One of the basic capabilities provided for tokens in ISR3 is file I/O. In particular, ISR3 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 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[3]. 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 will still write and read the pointers correctly, without getting into an infinite loop.

To use ISR3'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

[3]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.

use foreign file formats (see Section 2.2.2), the only formats a file will have are ISR3'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 reopening 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 **MatchFormatToFilename(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 2.2.2), so it creates code that can be applied to a wide variety of data.

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. The return type of Read_token_ptr() is **(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 2.3. 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. It can also be used to convert files to or from any of the foreign file formats. (This amazingly useful, albeit simple, utility is supplied as a stand-alone module in ISR3.)

2.2.2 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 image understanding systems, such as KBVision [8] or Khoros [6].

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 to ISR for these formats to understood. Viff files, for example, are read and written using the functions 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 this should not present a problem, since both environments are locally available.)

Table 2.1 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. Note that new foreign file formats can be (and are being) added to ISR3.

```
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 2.3: Example of File I/O: ISRconvert. 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 formats.

Extension	*Token Types*	*Host System*
.isra	all (ascii)	ISR3
.isrb	all (binary)	ISR3
.giff	ColorImage	Mosaic/PCs
.im	BytePlane, IntPlane, FloatPlane	KBVision
.tiff	ColorImage, BytePlane	Apple
.tks	TokenSets	KBVision
.viff	BytePlane, IntPlane, FloatPlane	Khoros

Table 2.1: File Formats currently recognized by ISR3.

Since the **MatchFormatToFilename** method of token streams knows about all of the file extensions in Table 2.1, the ISRconvert function shown in Figure 2.3 can be used to translate between compatible 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.

2.3 TokenSets

2.3.1 Abstract TokenSets

Among the basic functionalities provided by ISR3 are the abilities to group tokens into sets, logically manipulate sets of tokens, and access individual members and/or subsets of tokens. These facilities are provided through an abstract token type called the **TokenSet**. TokenSets provide functions for:

1. adding and removing tokens to/from a set.
2. taking the union, intersection and difference of sets.
3. accessing (subsets of) tokens by feature values.
4. iterating through the tokens in a set.
5. graphically displaying sets of tokens.
6. transforming the coordinate space of a set of tokens.

It is important to note that the TokenSet class is an abstract token type; it specifies the operations that are supported for a TokenSet, but not how the tokens are actually stored. ISR3 currently provides four subtypes of TokenSets, which store tokens in (variable length) arrays, linked lists, hash tables and two-dimensional grids. When a program creates a new TokenSet, it must specify which of these implementations is to be used (the choice should generally by made according to which data structure will be more efficient for a specific application). Many functions which use TokenSets can (and should) be written in terms of abstract TokenSets, however, so that they can be applied to data in any of the TokenSet formats.

Adding and Removing Tokens

Figure 2.4 shows the public interface to the TokenSet type. The first set of functions are the most basic. The **count()** method returns the number of tokens currently stored in the TokenSet. The **member(Token *)** method returns a positive integer if the argument Token is in the TokenSet, and -1 otherwise. (The value of the positive integer may or may not be meaningful, depending on the type of the TokenSet. TokenArrays, for example, return the index of the token, whereas TokenLists return either 1 or -1.) The **remove(Token *)** removes a token from a TokenSet.

The TokenSet class provides two methods for storing tokens in a TokenSet. The **add(Token *)** method first checks whether the argument token is already a member of the TokenSet, and only adds the token if it is new. This is important for guaranteeing that TokenSets are indeed sets (i.e. have at most one copy of any given Token), and consequently should be a programmer's default method of putting tokens into a TokenSet. The integer returned by add(Token *) is a positive number if the token was added to the TokenSet, and -1 if the token was already present. Since testing for membership may be expensive (depending in part on the TokenSet implementation) and there are times when a programmer knows that a token is not already part of a TokenSet, the **insert(Token *)** method can be used to insert a token into a TokenSet without first checking for membership. However, programmer's are warned that if a token is inserted into a TokenSet twice, subsequent operations (such as delete) may no longer work properly.

Logical Operations

Once tokens have been put into TokenSets, the logical operators **Union(TokenSet *)**, **Intersect(TokenSet *)** and **Diff(TokenSet *)** become useful. Programmer's should be aware that these operators are destructive to the sets they are applied to, so that given two TokenSets A & B, `A.Union(B)` will add to TokenSet A all the tokens in B (except those already in A). Similarly, `A.Intersect(B)` will remove from A any tokens that are not also in B, while `A.Diff(B)` will remove all the tokens in B from A.

Accessing Tokens

Putting Tokens into TokenSets would not be very useful if you could not access them later. ISR3 provides two basic mechanisms for accessing the Tokens in a TokenSet: a program can either iterate through the tokens in a TokenSet, as described in the next section, or it can retrieve specific tokens from a TokenSet, as described here.

Since a single TokenSet may contain many types of tokens, one way to retrieve tokens is by type. For example, if a program stored all the data extracted from an image in a single TokenSet, at some point it might want to access just the Line2D tokens; this is what the select methods are for. The **select(char *)** method takes the name of a token type as returned by its **name()** (e.g. "Line2D") method and returns the subset of tokens of that type. (Note that the returned value is itself a TokenSet.) The **select(int)** does the same thing, except that it uses the token type's index code as returned by the token's **type_index()** method to identify the token type. For example, assuming that TS and subTS are pointers to TokenSets, any of the following would select the Line2D tokens from a TokenSet `TS`:

- `subTS = TS->select(''Line2D'');`

```
class TokenSet : public Token {
 public:
  /* Addition and removal functions */
  virtual int count();
  virtual int member(Token *);
  virtual int remove(Token *);
  virtual int add(Token *);       /* checks for membership first  */
  virtual int insert(Token *);    /* doesn't check for membership */

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

  /* Access functions */
  TokenSet* select( char * );
  TokenSet* select( int );
  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 );

  /* Iteration functions */
  virtual TokenSet *new_inst();
  virtual TokenSetState* state();
  virtual TokenSetState* state(Token *);

  /* Coordinate system variables */
  CoordSys2D coords;
  CoordSys2D *coordinates() { return &coords; }
};
```

Figure 2.4: The public portion of the TokenSet declaration. (The basic token methods, such as name(), input() and Draw(), have been omitted for compactness sake.)

- `subTS = TS->select(ln.name());`
- `subTS = TS->select(ln.type_index());`

The last two forms require that the programmer already have access to a Line2D token called `ln`, which the first one does not.

More often, a programmer needs to access tokens by their feature values. This is done using one of the many forms of the **retrieve** method, depending on the type of the field to be retrieved on (int, float or char *) and whether or not all the tokens in the TokenSet are of the same type. The **retrieve(int offset, float min_value, float max_value)** is used to retrieve all tokens with a floating point field at location `offset` with a value between `min_value` and `max_value`, *assuming that all the tokens in the TokenSet are of the same type.* For example, given a TokenSet `TS` of Line2D tokens, a program could retrieve all the lines with a length between 10.0 and 20.0 with the command

- `TS.retrieve(offset_of(Line2D, Length), 10.0, 20.0)`

If the TokenSet `TS` includes tokens that are not Line2D tokens, then the **retrieve(char *tokentypename, int offset, float min_value, float max_value)** form of the retrieve method will skip other types of tokens, and return a TokenSet of just the Line2D tokens within the specified length range. In this case, the code would look like:

- `TS.retrieve(''line2D'', offset_of(Line2D, Length), 10.0, 20.0)`

Similar versions of the retrieve method exist for retrieving tokens based on the range of an integer field, namely **retrieve(int offset, int min_value, int max_value)** and **retrieve(char *tokentypename, int offset, int min_value, int max_value)**, again depending on whether or not there are multiple types of tokens within the TokenSet. With integer fields, it is also possible to retrieve tokens whose value exactly matches a target value with the methods **retrieve(int offset, int value)** and **retrieve(char *tokentypename, int offset, int value)**. Finally, tokens can be retrieved by an exact match of a field of type `(char *)` with the methods **retrieve(int offset, char *value)** and **retrieve(char *tokentypename, int offset, char *value)**.

Iteration

Perhaps the most common operation on a TokenSet is to iterate through the tokens it contains. To support iterating through arbitrary TokenSets, ISR3 introduces another (abstract) class, the **TokenSetState**. Conceptually, the TokenSetState represents the current position within a TokenSet, and it has two important methods. The **value()** method of a TokenSetState returns a pointer to the current Token in the TokenSet; the **next()** method advances the current position by one token and returns a pointer to the new token. Both methods return **NULL** once the state has advanced past the last token. (Figure 2.5 shows the public interface to the TokenSetState class.)

TokenSetStates are returned by the **state()** and **state(Token *)** methods of TokenSets. The `state()` method returns a TokenSetState that is initialized to the first token in the TokenSet; the `state(Token *)` method returns a TokenSetState that is initialized to point at the specified token, skipping any tokens that precede it.

Typically, TokenSetStates are used with the **for** statement to iterate through the elements of a TokenSet. For example, the Union operator for TokenSets can be implemented by the code shown

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

Figure 2.5: The public interface to the TokenSetState class.

```
TokenSet& TokenSet::Union( TokenSet *t2 )
{
  Token *token;
  TokenSetState *t2_state = c2->state();

  for ( token = t2_state->value(); token != NULL; token = t2_state->next() )
    add( token ); /* add() insert()s only if 'token' is not already there */

  delete t2_state;
  return *this;
} // TokenSet::Union( TokenSet * )
```

Figure 2.6: An example of iterating through a TokenSet. This Code shows the use of TokenSetStates and iteration to implement the TokenSet Union method.

in Figure 2.6; it iterates through the tokens in the source TokenSet, adding them to the destination TokenSet.

A more interesting example is code to compute some relation between all the pairs of tokens in a TokenSet. Assuming that the relationship is symmetric (so that if you've computed [A, B] you don't also want to compute [B, A]) and that you don't want to compare tokens to themselves, the code might like the code in Figure 2.7. This uses two TokenSetStates – one for the outer loop, iterating through every token, and the other for the inner loop, iterating through the tokens that the outer loop token has not yet been compared to.

(To avoid memory leeks, programmers should be aware that both versions of the **state** method create new TokenSetStates, and that these TokenSetStates should be deleted once they are no longer needed.)

2.3.2 TokenArrays

As mentioned earlier, TokenSet is an abstract token type. Although code can be written to manipulate generic TokenSets, whenever new TokenSets are created the program has to specify how the tokens should be stored. At the time of this writing, there are four options: TokenArrays, TokenLists, TokenHashTables and TokenGrids. All of these tokentypes are full implementations of the generic TokenSet type, and are therefore interchangeable for most purposes. Each type, however,

```
void compute_pairwise_relation(TokenSet *ts)
{
  TokenSetState *outer_loop = ts->state();
  TokenSetState *inner_loop;
  Token *tok1, *tok2;

  for(tok1 = outer_loop->value(); tok1 != NULL; tok1 = outer_loop->next()) {
    inner_loop = ts->state(tok1);
    for(tok2 = inner_loop->next(); tok2 != NULL; tok2 = inner_loop->next()) {
          .... pairwise relation code ...
    }
    delete inner_loop;
  }
  delete outer_loop;
}
```

Figure 2.7: Example of iterating through pairs of tokens using two TokenSetStates.

adds certain specific methods not available for all types of TokenSets (such as the spatial access methods of Grids). In addition, some operations are more efficient for some types of TokenSets than others.

TokenArrays store tokens in arrays. The non-negative numbers returned by the **add(Token *)**, **insert(Token *)**, **remove(Token *)** and **member(Token *)** methods of TokenArray are the index of the token in the array, and the

operator can be used to access indices in the array. Thus if `TA` is a TokenArray and `tok` is a token, both `TA[1] = tok` and `tok = TA[1]` are legal statements. TokenArrays can be sorted using either the **sort()** method, which sorts tokens according to their addresses in memory, or the **sort(int (*cmp)(Token *, Token *))** method, which sorts tokens according to a user-supplied function.

TokenArrays have no size bounds. If you add more tokens to a TokenArray than it currently can hold (or if you explicitly assign a token to a very large index), a new underlying array will be allocated that is large enough to hold the data, and the contents of the old underlying array will be copied into the new one (although none of this will be visible to the calling program). Thus it is not possible to generate an out-of-bounds error with a TokenArray, although it is not very efficient to start with a small array and then add large numbers of tokens to it.

The initial size of a TokenArray can be specified to the **new** command (i.e. in the constructor function; see Figure 2.8). If no initial size is given, an array with DEFAULT_ARRAY_SIZE[4] elements is created. A second argument can also be given specifying how much the TokenArray should grow by if it becomes overfull; the default is DEFAULT_ARRAY_INCREMENT[5]. To create an array of only fifty elements that will grow by only ten if becomes overfull, a program can specify `TokenArray`

[4] Currently, DEFAULT_ARRAY_SIZE is 100.

[5] Also currently 100.

```
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();
  int insert(Token *);
  int add(Token *);
  int remove(Token *);
  int member(Token *);
  TokenSet *new_inst();

  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 *) );
};
```

Figure 2.8: Public interface of TokenArray (remember that it inherits from TokenSet).

```
*TA = new TokenArray(50,10);.
```

2.3.3 TokenLists

TokenLists are an alternate implementation of TokenSets in terms of (singly) linked lists. TokenLists may be preferable to TokenArrays when the approximate size of a TokenSet is not known in advance, or if the only operation that is expected to be used on the TokenSet is iteration. (They tend to be less efficient otherwise.) Like TokenArrays, TokenLists can be sorted using either the **sort()** method, which sorts tokens according to their addresses in memory, or the **sort(int (*cmp)(Token *, Token *))** method, which sorts tokens according to a user-supplied function.

```
class TokenList : public TokenSet {
 public:
  TokenList();
  virtual ~TokenList();

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

  int count();
  int insert(Token *);
  int add(Token *);
  int remove(Token *);
  int member(Token *);
  TokenSet *new_inst();

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

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

Figure 2.9: Public interface for TokenList (remember that it inherits from TokenSet)

2.3.4 TokenHashTables

A third implementation of TokenSets is as TokenHashTables. By default, TokenHashTables hash tokens according to their address in memory. This has the advantage that it makes the **member(Token *)**, **add(Token *)** and **remove(Token *)** methods constant time, as opposed to $O(n)$ with TokenArrays and TokenLists. It also reduces the costs of **Union(TokenSet *)**, **Intersect(TokenSet *)** and **Diff(TokenSet *)** to $O(n)$ (from $O(n^2)$). The disadvantage is that the overhead for iterating through a TokenHashTable and for inserting (as opposed to adding) tokens is slightly higher than for other types of TokenSets.

The public declaration of a TokenHashTable is shown in Figure 2.10. The two special functions are **set_hash**, which replaces the default hash function with a user-specified one, and **set_compare_function**, which replaces the default equality test (whether or not the addresses are the same) with a user-specified one. Most users will not use either of these functions, although there are situations in which they may be useful. For example, one can imagine extracting line segments from an image by two independent methods, and being concerned that some line segments were extracted twice. Then one might store tokens in a TokenHashTable using a hash function that used the endpoints of the line segments as a key (perhaps multiplying the four endpoints together[6]).

2.3.5 TokenGrids

On the surface, TokenGrids are just like any other type of Tokenset; for example a user can add tokens to a TokenGrid, iterate through the tokens in a TokenGrid, or union two TokenGrids together. A TokenGrid is actually a two-dimensional array of smaller TokenSets, however, designed to provide spatial storage and retrieval for two-dimensional tokens.

In particular, a TokenGrid is a 2D array of TokenSets. Each TokenArray stores tokens that pass through a particular (rectangular) piece of the image. Between them, the TokenSets make up a tiling (or *grid*) that covers the image. When a token is added to a TokenGrid, it is conceptually rasterized to test which grid cells the token passes through, and the token is added to the corresponding TokenSets. This storage scheme, although redundant, allows special spatial retrieval functions which return the tokens in a TokenGrid passing through a particular point, circle or polygon in the image, or that intersect a particular line.

The tiling of the image is determined by the arguments to TokenGrid's initialization function. Although all the initialization arguments are optional, users may specify (in addition to the coordinate system) the size, starting point and increment of the tiling in each dimension. "Size" in this context refers to the size of the grid, in terms of how many grid cells there are in each dimension. The starting point refers to image coordinate at which the first grid cell begins, and the increment is the size of the individual grids cells (or the "increment" to the next boundary).

Figure 2.11 shows the TokenGrid initialization function. As an example of how it is used,

- `TokenGrid *grid = new TokenGrid(NULL, 64, 64, 0, 0, 4, 4);`

will create a grid which is a 128×128 array of TokenSets. Each TokenSet will cover a 4×4 patch of pixels, with the first TokenSet covering the range $[0, 4)$ in both the X and Y dimensions.

TokenGrids are not constrained to cover the area of any particular image, which is to say the $size \times inc$ does not have to equal the image dimension. TokenGrids may cover only part of an image,

[6] This is not a very good hash function, but it serves the pedagogical purpose.

```
class TokenHashTable : public TokenSet {
public:
  ~TokenHashTable();
  TokenHashTable (long n = DEFAULT_HASH_SIZE);
  char *name() {return "TokenHashTable";}

  int redundant_hash_p(Token *tok);             // Hash function returns unique values?
  void resize (long); // Resize for at least count
  inline long capacity () const; // Return max number of entries
  inline int is_empty () const;   // Determine empty/nonempty
  inline long get_bucket_count () const; // Return number of buckets
  inline void set_ratio (float); // Set growth ratio
  inline int get_count_in_bucket(long) const; // Used to return item count

  void statistics ();                           // Print Table Statistics
  void clear (); // Empty the hash table

  /* TokenSet virtual functions */
  int count () { return (int)entry_count;}  // Return number of entries
  int add (Token *tok);                         // Hash key/value
  int insert(Token *tok) {return add(tok);}
  int member (Token *tok);       // Get associated value for key
  int remove (Token *tok);         // Remove key/value from table
  TokenSet *new_inst() {return (TokenSet *) new TokenHashTable;}
  TokenSetState *state();                       // return TokenHashTableState
  TokenSetState *state(Token *tok);             // return TokenHashTableState

  int operator== (TokenHashTable&); // is equal
  inline int operator!= (const TokenHashTable&); // is not eq

  inline void set_hash (TokenHashTable_Hash); // Set hash function
  void set_token_compare (TokenHashTable_Token_Compare = NULL);
};
```

Figure 2.10: The public interface to TokenHashTable (inherits also from TokenSet)

```
#define DEFAULT_GRID_XINIT 0.0
#define DEFAULT_GRID_YINIT 0.0
#define DEFAULT_GRID_XINC 4.0
#define DEFAULT_GRID_YINC 4.0
#define DEFAULT_XSIZE 128
#define DEFAULT_YSIZE 128

public:
  TokenGrid( CoordSys2D *coord_sys = NULL,
        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);
```

Figure 2.11: The TokenGrid initialization function declaration (note that all arguments are optional). Size refers to the number of grid cells in a dimension, init refers to the image coordinate that the first cell starts at, and inc refers to the size of an individual cell.

of they may extend beyond the boundaries of the image. Indeed, there is no tie between a TokenGrid an any particular image, and it is possible to store tokens from many different images (of different sizes) in a single TokenGrid. (Obviously, token coordinates must be expressed in some coordinate system, and may have to be transformed to match the coordinate system of the TokenGrid. We leave this discussion to Section 2.4.)

In the example above, the TokenGrid was defined to cover a 128×128 image. It could be extended to cover a 256×256 image either by doubling the number of grid cells in each dimension (arguments 2 and 3), or by doubling the size of the individual cells (arguments 6 and 7). The choice affects only efficiency, and depends on the density of tokens in image space and the spatial size of the average retrieval request.

If a token lies outside the spatial range of a TokenGrid (i.e. has coordinates less than init or greater than $init + (size * inc)$), then it will not be put in any grid cell and cannot be retrieved spatially, although it is still "in" the TokenGrid (e.g. member() still returns a positive number[7]). In general, it is a good idea to make a TokenGrid big enough to cover all the tokens that are to be put in it. It should be noted in this context that negative coordinates are not a problem for TokenGrids. Some tokens (such as intersections between lines) may be off the image plane, but these can easily be stored in TokenGrids by using negative values for x_init and y_init.

Figure 2.12 shows the additional retrieval functions provided for TokenGrids. In general, there is the same distinction between retrieval and selection functions as with TokenSets: retrieve functions assume that all tokens in the Grid are of one type, while select functions check the type of each token before they retrieve it, selecting only those that are of a specified type. With these functions, however, retrieval is based on the (2D) location of a token, rather than on the value of a single field.

The most commonly used retrieval function for TokenGrids is **retrievePolygon** (and its cousin, **selectPolygon**). RetrievePolygon retrieves any token in TokenGrid that intersects an arbitrary polygon, as specified by a list of 2D points. Sometimes, the polygon is a scan-line aligned rectangle,

[7]Tokens with no geometric extent, such as Transform2D tokens, will suffer the same fate.

```
public:
   TokenSet *retrievePoint(CoordSys2D *csys, double x0, double y0);
   TokenSet *selectPoint(CoordSys2D *csys, char *type,
                         double x0, double y0);
   TokenSet *retrieveLine(CoordSys2D *csys, double x1, double y1,
                                            double x2, double y2);
   TokenSet *selectLine(CoordSys2D *csys, char *type, double x1,
                        double y1, double x2, double y2);
   TokenSet *retrieveRectangle(CoordSys2D *csys, double x1, double y1,
                                                 double x2, double y2);
   TokenSet *selectRectangle(CoordSys2D *csys, char *type, double x1,
                         double y1, double x2, double y2);
   TokenSet *retrieveCircle(CoordSys2D *csys, double x0, double y0,
                         double radius);
   TokenSet *selectCircle(CoordSys2D *csys, char *type, double x0,
                         double y0, double radius);
   TokenSet *retrievePolygon(CoordSys2D *csys, Point2D points_list[MAX_POINTS],
                         int num_of_points);
   TokenSet *selectPolygon(CoordSys2D *csys, char *type,
                         Point2D points_list[MAX_POINTS],
                         int num_of_points);
```

Figure 2.12: TokenGrid retrieval functions. In addition to the retrieval functions inherited from TokenSet, TokenGrids provide these spatial query functions.

in which case **retrieveRectangle** (or **selectRectangle**) will be faster. At other times, the search is for tokens within some distance of a point, in which case **retrieveCircle** (or **selectCircle**) is appropriate, with the point being the center of the circle (i.e. x0, y0) and the distance threshold being the radius. It is also possible to retrieve the tokens intersecting (or overlapping) a single point or a line, although this seems less common in practice.

2.4 Coordinates

Until Section 2.3.5, we were able to describe ISR3 without mentioning the term *coordinate system.* After all, ISR3 manipulates symbolic objects by writing them to files and reading them back again, or by storing them in sets. It does not, in general, care about the semantics of the fields of the tokens. Visual tokens are different from other objects, however, in that many of them are geometric objects, and when ISR3 exploits this fact to provide spatial access facilities it is forced to address the question of what coordinate system the data is expressed in.

Coordinate systems are of course fundamental to visual data. ISR3's graphics facilities (described elsewhere) depend on knowing the coordinate system of the data. Moreover, even when ISR3 itself doesn't need to know the coordinate system, the user surely does – what good is a set of line segments saved in a file, if the user doesn't know what coordinate system the endpoints are stored in? Clearly, representing coordinate systems and transforming between coordinate systems is a necessary function of a visual database system.

Ultimately, any coordinate system is as good as any other (although coordinate systems with orthogonal bases are generally preferred), and ISR3 allows tokens to be expressed in any coordinate system the user chooses. What ISR3 provides is the ability to test whether any two tokens are in the same coordinate system, and if not to transform one into the coordinate system of the other. For two-dimensional tokens this ability is provided by two classes, the **CoordSys2D** token type and the **Transform2D** token type. (The same capabilities will soon be provided for 3D objects through CoordSys3D and Transform3D token types, but these are not yet implemented.)

2.4.1 Transform2D

Coordinate systems have no "absolute" definition; they are defined only in terms of the transformations that map points in one coordinate system onto points in another. These transforms can divided into component parts (e.g. rotation, translation, scale, shear), but they are most easily thought of in terms of a single transformation matrix. In particular, ISR3's **Transform2D** token type models a 2D coordinate transformation as a 3×3 matrix mapping 2D points in homogeneous coordinates to 2D points in homogeneous coordinates.

For readers not familiar with homogeneous coordinates, they are technique which adds an extra dimension to the representation of a point so as to cast many common operations as linear operations as reduce the number of special case conditions. In homogeneous coordinates, the point [x, y] is most commonly represented by the three-place vector [x, y, 1]. The last coordinate is actually a scale term, so the vector [2x, 2y, 2] represents the same point in space as [x, y, 1], implying that the representation of points is not unique. (The vector [x, y, 0] specifies a point at infinity in the direction of [x, y].) The payoff for sacrificing uniqueness, however, is that coordinate transformations can be reduced to a single matrix multiplication. To rotate, translate and scale one coordinate system

```
class Transform2D : public Token {
  friend Transform2D* operator* (Transform2D& t1, Transform2D& t2);
 public:
  Transform2D();   /* returns identity transform */
  Transform2D(double r11, double r12, double tx,
      double r21, double r22, double ty,
      double a0 = 0.0,  double a1 = 0.0,  double s = 1.0);
  Transform2D(Transform2D* ts);

  void output(Isrostream&);
  void input(Isristream&);
  char *name() {return "Transform2D";}

  double Determinant();
  double Minor(int, int);
  Transform2D *Inverse();
  void Apply(double x, double y, double s, double &ux, double &uy, double &us);
  int Apply(double x, double y, double &ux, double &uy);
  int operator== (const Transform2D& other);
  inline int operator!= (const Transform2D& other)
{return (!operator== (other));}
};
```

Figure 2.13: The public interface to the Transform2D token type.

relative to another, for example, simply multiple points in the first system by:

$$\begin{bmatrix} cos & sin & tx \\ -sin & cos & ty \\ 0 & 0 & scale \end{bmatrix}$$

where cos and sin are the obvious trigonometric functions of the angle of rotation, (tx, ty) is the translation vector and scale is the relative scale. (The signs on the sin term may depend on whether the coordinate system is left-handed or right-handed. Shear can be introduced using matrix elements 3,1 and 3,2, but that is beyond the scope of this manual.) As the reader will see, however, users do not have to use homogeneous coordinates just because ISR3 does.

Figure 2.13 shows the public interface to the Transform2D token type. To create a new Transform2D, a user should specify the terms of the 3×3 homogeneous transformation matrix; fortunately, as will be discussed in the next section, most users will construct Transform2Ds by overlaying two existing coordinate systems, not by building them explicitly. Creating a Transform2D without supplying any arguments creates a new instance of the identity transformation.

Transform2Ds have the properties you would expect of a square matrix: they can be multiplied together[8], tested for equality, and their inverses, determinants and minors can be computed. Most

[8]Users concerned about memory leaks should be aware that multiplying to Transform2Ds together creates a new

importantly, they can be used to transform points from one coordinate system to another. For user's who are comfortable with homogeneous coordinates, the **Apply(double x, double y, double s, double &ux, double &uy, double &us)** method takes a 2D point as a three-place homogeneous vector and returns the transformed coordinates in the same format (through the call-by-reference variables). The Apply method will always set the return value **us** to be either 0 or 1. For users who prefer to use just two coordinates to represent a 2D point, the **Apply(double x, double y, double &ux, double &uy)** does the same thing without worrying about homogeneous coordinates. The major difference is that this transformation can fail (trying to divide by zero), so this version of the Apply method returns an integer that is 1 if the multiplication succeeded and zero otherwise.

2.4.2 CoordSys2D

Although important themselves, the primary role of Transform2Ds is to define the relationship between two coordinate systems. In ISR3, two-dimensional coordinate systems are represented by tokens of type **CoordSys2D**. *Every ISR3 token inherits a private field called* ***coord*** *that is a pointer to CoordSys2D token,* and can be accessed by a **GetCoordinates()** method inherited from the Token class. If a token is not a 2D geometric token, this field should be left NULL. Otherwise, it should be set to the proper coordinate system using the **SetCoordinates(CoordSys2D *sys)** method (also inherited from the Token class).

How does a user create an appropriate CoordSys2D token? The easiest way is to adopt or modify one of ISR3's default coordinate systems. ISR3 predefines five coordinate systems for users. The first is called **UNIT_XWINDOW**, and it is a unit version of the coordinate system used by Khoros [6] and XWINDOWS. That is to say, the origin is in the upper left-hand corner of the window, with x increasing to the right and y increasing down. It is a *unit* coordinate system because it is based on a mythical window that goes from 0 to 1 in each dimension (although arbitrarily large coordinates are allowed).

The other four default coordinate systems are also unit systems. **UNIT_XY** puts the origin at the bottom left of the (mythical) window, with increasing to the right and y increasing up. (This is the unit version of most math text coordinate systems as well as KBVision [8].) **UNIT_ROWCOL** is like UNIT_XWINDOW except that the first coordinate (row) increases as it goes down while the second coordinate (col) increases to the right. (This is commonly used in graphics.) **UNIT_MID_X** and **UNIT_MID_Y** put the origin in the middle (with each dimension of the mythical window going from -0.5 to 0.5), with the first dimension increasing to the right. The difference between them is that the left-handed version has the second dimension increasing upward, while the right-handed version has the opposite.

The default coordinate systems are all unit coordinate systems so that users can scale them to the appropriate size (usually, but not always, determined by image size). Scaled coordinate systems are made by applying the **Scale(double s)** or **Scale(double sx, double sy)** method to an existing coordinate system (often one of the defaults). Users can make other coordinate systems by applying the **Translate(double tx, double ty)** or **Rotate(double rot, RotationUnit measure)** methods to an existing CoordSys2D, or by creating a new CoordSys2D from an arbitrary Transform2D token and an existing CoordSys2D.

So what can you do with a CoordSys2D once you have it? Not a lot. Since every token is ideally

---

Transform2D.

```
class CoordSys2D : public Token {
  friend Transform2D* overlay(CoordSys2D* source, CoordSys2D* dest);
 public:
  /* create base (identity) coordinate system */
  CoordSys2D();
  /* define new coordinate system in terms of a transform that converts
     it to any known base coordinate system */
  CoordSys2D(Transform2D *trans, CoordSys2D *base);
  CoordSys2D(CoordSys2D *sys);
  ~CoordSys2D() { delete sys2base; delete base2sys; }

  inline int operator== (const CoordSys2D& other)
   {return *sys2base == *(other.sys2base);}
  inline int operator!= (const CoordSys2D& other)
{return (!operator== (other));}
  CoordSys2D* Translate(double DX,double DY);
  CoordSys2D* Scale(double SC);
  CoordSys2D* Scale(double xscale, double yscale);
  CoordSys2D* Rotate(double amount, RotationUnit measure);
};
```

Figure 2.14: The public interface to the CoordSys2D token type. (Some functions inherited from Token have been deleted for compactness.)

in some coordinate system, we didn't want to pay the overhead of storing coordinate transformation data in each. Instead, each token has a pointer to a CoordSys2D, and some CoordSys2Ds will have many tokens that point to them. We therefore had to be careful about letting users side-effect CoordSys2D tokens.

As shown in Figure 2.14, the public interface to CoordSys2D allows a user to make a new CoordSys2D as a copy or transformation of an existing one, and to test whether two CoordSys2Ds are equal. The special function **Transform2D *overlay(CoordSys2D *source, CoordSys2D *dest)** creates the Transform2D token that will map coordinates in one coordinate system (called source) to points in another (called destination). It is also possible to delete a CoordSys2D, but user's should not do this unless they know that no token has a pointer to it.

In general, users should do two things with CoordSys2D tokens: 1) create them, usually by scaling, translating and rotating existing coordinate systems as already discussed, and 2) set the coordinates of tokens to be in a given CoordSys2D. Most 2D geometric tokens will have a NULL coordinate system when they are created. By specifying a coordinate system, a user tells the system (and future users) the intended coordinate system.

The **SetCoorinates(CoordSys2D *sys)** method, inherited from the Token class by all 2D geometric tokens, should be used to set the coordinate system of a token. If the coordinate system of the token was previously NULL, it will be set to the new CoordSys2D. If the coordinate system

of the token was a different CoordSys2D, then *the coordinates of the token will be destructively transformed to the new coordinate system.* How these tokens are transformed depends upon their type: Line2D tokens, for example, are transformed by changing their endpoints, while Edgel tokens are transformed by changing their midpoint and orientation. Note that these changes are destructive, however. (Resetting a token to its current coordinate system is a no-op; setting the coordinate system of a non-geometric token is an error.)

Chapter 3

Programmer's Reference

ISR3 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**. The *raison-detre* of this chapter is that inheriting from Token is a necessary but not sufficient condition for being an ISR3 token; certain virtual methods have to be defined as well. The next section, Section 3.1, 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.

3.1 Simple Tokens

In order for a C++ class to qualify as a token type, 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 3.1 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's initial classes; the full definition will be shown later (it was also shown earlier in Figure 2.1). Figure 3.1 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.

The second requirement for making a class into an ISR3 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 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 3.2. If multiple token classes are defined in a single file, each class should be given its own entry.

3.2 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++.

```
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 3.1: A simplified version of the Line2D token definition showing the name method definition.

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

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 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++.

3.2.1 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. This is acceptable (although rarely desirable) in some applications, but not in applications such as ISR3 which rely on generic operations. In particular, ISR3 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 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*[1].

3.2.2 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.

[1] 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*.

Unfortunately, this only solves half the problem. If a class has multiple (virtual) parents that (virtually) inherit *Token*, then ISR3 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.[2] This leads to the second restriction that *ISR3 classes cannot inherit* Token *as a virtual base class.*

3.3 Coordinates

If a programmer is designing a two-dimensional, geometric token class, then that class should respond correctly to coordinate transformations, and should be rasterizable (if thats a word) so that it can be stored in TokenGrids. The first requires defining a method called **SetCoordinates**, which takes a pointer to a CoordSys2D as its argument. If the coord field of the token is NULL (remembering that all tokens inherit a coord field from the Token class), then the SetCoordinates method should simply set the coord field to the address of its argument. If the coord field is not NULL, however, it must destructively transform the token to put it in the new coordinate system.

Figure 3.3 shows the SetCoordinates method for Line2D tokens. If coord was already a pointer to the new CoordSys2D, then SetCoordinates is a no-op; if coord was previously NULL then it is simply set to the new CoordSys2D. Otherwise, things get more complicated.

In particular, it has to translate the endpoints of the line segment to the new coordinate system. This is accomplished by using **overlay** to compute the Transform2D between the original coordinate system and the new one, and then calling the **Apply** method of Transform2D to calculate the new positions of the endpoints.

An annoying but important detail is the presence of the second and third arguments. It is fairly rare for a program to change the coordinates of one token at a time. Typically a program will change the coordinate system of a TokenSet of tokens, or of an entire subtree of data. In this case, recomputing the transformation from the old coordinate system to the new one for every token would be very expensive. Instead, tokens that group other tokens (such as all of the TokenSet classes) compute the transform from their old coordinate system to the new one, and the old coordinate system and the precomputed transform are passed in as the second and third arguments to SetCoordinates. If the new coordinate system does not match a token's original coordinate system, the SetCoordinates routine should check its second argument. If this is the same as the token's old coordinate system, then it does not have to recompute the transformation using overlay(); it can use the Transform2D passed in as its third argument instead.

On the flip side, it is important for the SetCoordinates() method to compute these extra parameters for token types that group other tokens. For example, Figure 3.4 shows the SetCoordinates method for the TokenSet class.

[2]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 (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.

```
void Line2D::SetCoordinates(CoordSys2D *new_sys, CoordSys2D *old_sys,
    Transform2D *old_trans)
{
  double tx1, tx2, ty1, ty2, dx, dy;
  Transform2D *trans;

  if (coord == new_sys) return;
  if (coord == NULL) {
    coord = new_sys;
    return;
  }

  if (coord == old_sys) trans = old_trans;
  else trans = overlay(coord, new_sys);

  if (trans != NULL) {
    if (trans->Apply(x1, y1, tx1, ty1)) {
      x1 = tx1;
      y1 = ty1;
    }
    if (trans->Apply(x2, y2, tx2, ty2)) {
      x2 = tx2;
      y2 = ty2;
    }
    dx = x2-x1;
    dy = y2-y1;
    theta = atan2(dy, dx);
    length = sqrt((dx * dx) + (dy * dy));
  }
  coord = new_sys;
  if (trans != old_trans) delete trans;
}
```

Figure 3.3: The Line2D SetCoordinates method.

```
void TokenSet::SetCoordinates(CoordSys2D *new_sys, CoordSys2D *old_sys,
      Transform2D *old_trans)
{
  Transform2D *trans = old_trans;
  TokenSetState *cs = state();
  Token *tok;

  if ((coord != NULL) && (coord != old_sys) && (new_sys != coord)) {
    old_sys = coord;
    trans = overlay(coord, new_sys);
  }

  for (tok = cs->value(); tok != NULL; tok = cs->next())
    tok->SetCoordinates(new_sys, old_sys, trans);

  delete cs;
  coord = new_sys;
  if (trans != old_trans) delete trans;
}
```

Figure 3.4: The TokenSet SetCoordinates method.

3.4 File I/O

One of the basic capabilities provided for tokens in ISR3 is file I/O. In particular, ISR3 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. Unlike many other visual representation systems, ISR3 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 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 that the current source token has a pointer to. The default definition for trace is a no-op, so token classes that do not contain pointers to 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's native data formats; they should not be manipulated at the level of applications programs (use TokenOStream and TokenIStream for this purpose). In most ways, Isrostreams are like standard C++ ostreams[3], 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

[3]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.

```
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 3.5: 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 preceded 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.

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 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 3.5 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.

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.

```
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 3.6: 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.

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 3.6, which shows the input function for the Line2D class. The only significant difference between the bodies of code in Figure 3.5 and Figure 3.6 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 it. We can therefore expand the Line2D token definition shown in Figure 3.1 to the one in Figure 3.7 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 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 [7]. As shown in Figure 3.8, 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 3.9.

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 tokens 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 3.10 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 guarantees 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 separate tokens with the same feature values.

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

3.5 Graphics Methods

One of the most important facilities ISR3 provides users is graphics. ISR3 includes a stand-alone executable graphics program called **ISRWish**[3] that reads and graphically displays the contents of data files. **ISRWish** is an extension of **Tcl/Tk**[5]. Also included is a **Tcl/Tk** script named **xisrdisplay**[4]. It allows users to examine data [4], overlay data, and interactively manipulate data with a mouse. By far the most important feature of **ISRWish** and **xisrdisplay**, however, are their 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.

3.5.1 Generic Graphics Device Class

Both *Draw()* and *BitDraw()* are passed a *Generic Graphics Device Class* instance. This object defines a set of methods for performing graphical output operations, and is defined in Figure 3.11. ISR3 currently supports two types of graphical output: XWindow output and PostScript output. The Generic Graphics Device Class implements both generically, so that Draw() and BitDraw() can be written to produce either form of output. Of course, users are free to implement additional device drivers, but that is outside the scope of this manual.

This class is device-independent, but when the *Draw()* or *BitDraw()* methods are actually called, a class derived from this class will be passed, a class that has device-dependent methods for the device in use (X11 and Tk PostScript classes have been written and are included in **ISRWish**).

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

```
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 3.8: 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 3.9: The trace() function definition for Line2DPair. The trace function tells ISR3 which tokens a Line2DPair points to.

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

3.5.2 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 *raster* tokens such as images that are displayed in terms of pixel images. For vector tokens class designers should provide a *Draw()* method, while for raster tokens a *BitDraw()* method should be defined. 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 six arguments and returns void (as shown in Figure 3.12), but draws the token to the graphics device as a side-effect. The first argument is a reference to a *GraphicsDevice* instance. Since **ISRWish** allows users to interactively set the window size, zoom factor and graphics context there is no reason for the Draw() method to alter the graphics context represented by the GraphicsDevice instance. The second 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 its CoordSys2D object; this may or may not match the coordinate system of the window. Moreover, the mapping from token coordinates to window coordinates changes interactively as the user zooms and roams around the image. The second 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

```
class GraphicsDevice {
public:
        virtual void DrawPoint(double x,double y) = 0;
        virtual void DrawLine(double x1,double y1,double x2,double y2) = 0;
        virtual void DrawArc(double  x,double y,double w,double h,
                             double a1,double a2) = 0;
        virtual void DrawEllipse(double a,double b,double p,double x0,
                                 double y0) = 0;
        virtual void DrawRect(double  x,double y,double w,double h) = 0;
        virtual void FillArc(double  x,double y,double w,double h,
                             double a1,double a2) = 0;
        virtual void FillEllipse(double a,double b,double p,double x0,
                                 double y0) = 0;
        virtual void FillRect(double  x,double y,double w,double h) = 0;
        virtual int  SetNextColor(double r,double g,double b) = 0;
        virtual int  DisplayDepth() = 0;
        virtual bool IsColor() = 0;
        virtual void SetCurrentColor(int icolor) = 0;
        virtual bool AllocateImage(int width,int height, int depth) = 0;
        virtual bool ImageIsAllocatedP(int& width,int& height,
                                       int& depth) = 0;
        virtual bool DeAllocateImage() = 0;
        virtual bool ReAllocateImage(int width,int height, int depth) = 0;
        virtual int  GetImagePixel(int x,int y) = 0;
        virtual void PutImagePixel(int x,int y,int value) = 0;
        virtual void FillImageRect(int x,int y,int w,int h,int val) = 0;
        virtual void DrawImage(double destx,double desty,int srcx,int srcy,
                               int w,int h) = 0;
};
```

Figure 3.11: The GraphicsDevice class.

```
/*--------------------------------------
  gd       -- GraphicsDevice object
  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(GraphicsDevice& gd,Transform2D *trans,
                 double xmin, double ymin, double xmax, double ymax);
```

Figure 3.12: The Draw method for vector graphics. The first argument is the GraphicsDevice instance, the second defines the token to window coordinate transformation, and the last four specify the area being (re)drawn.

can be used in the GraphicsDevice draw methods, as shown in Figure 3.13.

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 3.13 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.

3.5.3 Raster Tokens

Some tokens, such as images, should be drawn not by making repeated calls to primitive draw routines, but by creating and displaying a pixel image. 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 **ISRWish** draws a token. Drawing a pixel image requires additional information about the depth of the display window (in bits) and the size of window being drawn to. This information is supplied by the GraphicsDevice (see Figure 3.11). BitDraw therefore has the same arguments as Draw, as shown in Figure 3.14.

Internally, BitDraw() methods work by using the AllocateImage() and DrawImage() methods of the GraphicsDevice instance. It can be generally presumed that for a given display instance (i.e. a given display window), the GraphicsDevice class instance will be re-used everytime the object needs to be (re-)drawn. It is only necessary to call the AllocateImage() once and only necessary to call ReAllocateImage() when the window size or zoom factor changes – which can be tested by re-computing the image size (using the Apply() method of the transform on the extents of the object).

The BitDraw() method for Plane is shown in Figures 3.15, 3.16, and 3.17. First the logical image size is computed (Figure 3.15). This is the size (in device units) the Plane will be after being

```
void Line2D::Draw(GraphicsDevice& gd, Transform2D* trans,
                  float xmin, float ymin, float xmax, float ymax)
{
  double pts1[2], pts2[2];

  /* 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 */
  trans->Apply(x1, y1, pts1[0], pts1[1]);
  trans->Apply(x2, y2, pts2[0], pts2[1]);

  /* draw line */
  gd.DrawLine(pts1[0], pts1[1], pts2[0], pts2[1]);

}
```

Figure 3.13: 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.

```
/*----------------------------------------
   gd       -- GraphicsDevice object
   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
----------------------------------------*/

void Token::BitDraw(GraphicsDevice& gd, Transform2D *trans,
                    double ulx, double uly,
                    double lrx, double lry);
```

Figure 3.14: The BitDraw method for vector tokens. The arguments are the same as for Draw(), except that the re-display coordinates are spacially sorted, not numerically sorted.

transformed. Then the GraphicsDevice is checked to see if its internal image has been allocated and is the correct size. If the internal image is the wrong size (the window or zoom factor size changed or this is the first time), the internal image is (re-)allocated (Figure 3.16) and the plane is (re-)drawn into the image (Figure 3.17). Finally, the portion of the image that needs redisplay is redisplayed with the DrawImage() method of the GraphicsDevice instance (also in Figure 3.17).

3.5.4 Non-graphical Tokens

Although the preceding 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 necessary for every token to produce graphical output, although most tokens should.

```
void Plane::BitDraw(GraphicsDevice &gd,Transform2D* trans, float xmin,
                    float ymin, float xmax, float ymax)
{
  /* figure displayed image size */
  double wsizex, wsizey, wzerox, wzeroy;
  trans->Apply((double)XSize,(double)YSize,wsizex,wsizey),
  trans->Apply(0.0,0.0,wzerox,wzeroy);
  double winwidth = fabs(wsizex-wzerox), winheight = fabs(wsizey-wzeroy);
  /* figure re-draw area */
  double c1x, c1y, c2x, c2y;
  trans->Apply(xmin,ymin,c1x,c1y); trans->Apply(xmax,ymax,c2x,c2y);
  double imw = fabs(c1x-c2x), imh = fabs(c1y-c2y);
  /* figure Plane pixel to GraphicsDevice pixel mapping */
  double hpix = (int) ((winwidth / (double) XSize) + 0.5),
         vpix = (int) ((winheight / (double) YSize) + 0.5);
  int delta_r = 1, delta_c = 1;
  if (winwidth < XSize)
  { hpix = 1; delta_c = (int)((XSize / (double) winwidth) + 0.5); }
  if (winheight < YSize)
  { vpix = 1; delta_r = (int)((YSize / (double) winheight) + 0.5); }
  /* compute source image origin */
  double isrcx = (xmin * hpix) / delta_c, isrcy = (ymin * vpix) / delta_r;
  if ((imw+isrcx) > winwidth) imw = winwidth-isrcx;
  if ((imh+isrcy) > winheight) imh = winheight-isrcy;
  /* displayed image too small to see? */
  if (imw <= 0 || imh <= 0) return;
  /* integerize image size */
  int ImWidth = (int)(winwidth+0.5), ImHeight = (int)(winheight+0.5);
```

Figure 3.15: The BitDraw method of Plane (size and coordinate computations). See text for a complete description.

```
  /* get display depth */
  int depth = gd.DisplayDepth();
  /* now check to see if the display image has been already
     allocated and if it is the right size. */
  int oldWidth, oldHeight,oldDepth;
  bool ReDraw;
  if (!gd.ImageIsAllocatedP(oldWidth, oldHeight,oldDepth))
  {
    if (!gd.AllocateImage(ImWidth,ImHeight,depth))
    { cerr << "Out of memory in NewPlane::BitDraw\n"; return;
    } else ReDraw = true;
  } else if (oldWidth != ImWidth || oldHeight != ImHeight ||
             oldDepth != depth)
  {
    if (!gd.ReAllocateImage(ImWidth,ImHeight,depth))
    { cerr << "Out of memory in NewPlane::BitDraw\n"; return;
    } else ReDraw = true;
  } else ReDraw = false;
```

Figure 3.16: The BitDraw method of Plane (GraphicsDevice internal image check). See text for a complete description.

```
  /* does the image need re-drawing? */
  if (ReDraw)
  {
    if (depth == 1) ditherBitImage(gd,ImWidth,ImHeight,
                                   (int)(hpix+.5),(int)(vpix+0.5),
                                   delta_r,delta_c);
    else  drawGreyImage(gd,ImWidth,ImHeight,(int)(hpix+.5),
                        (int)(vpix+0.5),delta_r,delta_c);
  }
  /* update screen (or whatever) */
  gd.DrawImage(c1x,c1y,(int)(isrcx+0.5),(int)(isrcy+0.5),
               (int)(imw+0.5),(int)(imh+0.5));
}
```

Figure 3.17: The BitDraw method of Plane (Actual drawing code). See text for a complete description.

Bibliography

[1] John Brolio, Bruce A. Draper, J. Ross Beveridge and Allen R. Hanson. "The ISR: an intermediate-level database for computer vision", *Computer*, 22(12):22–30 (Dec. 1989).

[2] Bruce A. Draper, Gökhan Kutlu, Edward M. Riseman and Allen R. Hanson. "ISR3: Communication and Data Storage for an Unmanned Ground Vehicle," Proc. of the 12th Int. Conf. on Pattern Recognition, Jerusalem, Oct. 1994, Vol 1, pp. 836–838.

[3] Robert Heller. "Program Notes – how to write your own GUI using ISRWish", June 1995. Available here in the same directory.

[4] Robert Heller. "Xisrdisplay - A Tcl/Tk Program For Displaying ISR Token Objects", June 1995. Available here in the same directory.

[5] John K. Ousterhout. *Tcl and the Tk Toolkit*, Reading, MA., 1994 (Addison-Wesley Publishing Company).

[6] John Rasure and Steven Kubica. "The Khoros Application Development Environment," in *Experimental Environments for Computer Vision and Image Processing,* H.I. Christensen and J.L. Crowley (eds.), World Scientific Press, Singapore, 1994. pp. 1–32.

[7] George Reynolds and J. Ross Beveridge. "Searching for Geometric Structure in Images of Natural Scenes," *Proc. of the DARPA Image Understanding Workshop,* Los Angeles, CA., Feb. 1987. pp. 257–271 (Morgan-Kaufman Publishers, Inc.).

[8] Tom Williams. "Image Understanding Tools,", *Proc. of the 10th Int. Conf. on Pattern Recogntion,* Atlantic City, NJ, June 1990. pp. 606–610.