International Journal of Computational Geometry & Applications

GEOSHEET: A DISTRIBUTED VISUALIZATION TOOL FOR GEOMETRIC ALGORITHMS*

D.T. LEE† CHIN-FANG SHEN‡ DENNIS S. SHEU§

Department of Electrical and Computer Engineering
Northwestern University
Evanston, Illinois 60208, USA.
E-mail:{dtlee,cfshen,sheusm}@ece.nwu.edu

Received 13 April 1995
Revised 2 April 1997
Communicated by N. Amenta and S. J. Fortune

ABSTRACT

GeoSheet (version 1.0) is an interactive visualization tool for visualizing geometric algorithms in distributed environments. It provides features such as interactive visualization of program states for debugging, high-level graphical input/output manipulation facilities for geometric objects, reuse of existing data structures and algorithms implementation, and more importantly distributed executions on heterogeneous machines at different sites. To minimize development effort of the tool we make use of existing software packages available in public domain. Specifically we extend Xfig with a message-driven interface and a socket-based interprocess communication (IPC) mechanism. This *extended*-Xfig is the backbone of this version of the tool. Object-oriented programming methodology is used to construct the visualization interface. By deriving from traditional data type and algorithm libraries, our abstract GeoObject representation super-classes are easy to use, easy to construct, and highly portable. Although *GeoSheet* is not restricted to a particular application domain or any programming language, this release only contains geometric algorithm implementations in C++ and LEDA. We hope that the geometric algorithm designers will find it useful when they develop their algorithms.

Keywords: Visualization, geometric computing, objected-oriented programming, X windows, distributed environment.

1. Introduction

Using conventional approaches it is difficult to understand the structures of geometric objects represented in numerical format. Perception of objects is better obtained from visualization, as graphical form is most natural to human eyes, and conveys more information than numerical values or texts. By the same token to better understand the behavior of an algorithm, be it geometric or not, we would

*Supported by the Office of Naval Research under the Grant No. N00014-93-1-0272.
†Research also supported by the National Science Foundation under the Grant CCR-9309743.
‡Current address: Avant! Corporation, 46871 Bayside Parkway, Fremont, CA 94538.
§Current address: ZEXEL Innovation, 1249 Innsbruck Dr., Sunnyvale, CA 94089.

make use of visualization. Techniques and tools for algorithm visualization are thus developed in hopes that they not only help describe the algorithm behavior after they are fully implemented, but also help algorithm developers debug the programs when they are being implemented. The tools described in this article were developed primarily for geometric algorithm designers who often deal with both geometric and graph objects, and for educators who teach algorithms to non-experts such as students and application engineers. In this release, *GeoSheet* supports 2-dimensional geometric algorithm visualization. It is written in C++ running under UNIX and X windows environment.

To produce graphical output, the user is usually required to have special and non-trivial domain knowledge on visualization programming, such as computer graphics and animation. Such domain knowledge cannot be quickly acquired. Although many systems are now available for algorithm visualization,[18,24,30] users of most systems still face problems in using the visualization tools when implementing geometric algorithms. To invoke visualization functions provided in these systems, the user oftentimes has to follow specific instructions to place special statements within programs. These instructions are usually restricted, and tedious to follow, rendering the tools hard to use. Furthermore, additions of these special visualization commands in conventional Pascal or C programs will clutter the program flow and reduce its readability.

In view of the above shortcomings we adopt object-oriented approach to develop a tool, called *GeoSheet*, that can be used with minimal knowledge of the visualization programming, and in which the invocation of the visualization functions is done in a manner similar to the conventional *printf* and *scanf* commands of C.

GeoSheet is an interactive visualization tool designed to simplify geometric algorithms visualization procedures in a distributed environment. It is display device independent, although in the current release, the display is based on Xfig (Facility for Interactive Generation of Figures under X11) and is primarily for 2-dimensional objects. This tool is developed as part of a bigger project, GeoMAMOS (for Geometric object MAnipulation and MOnitoring System), supported by the Office of Naval Research.[a]

To visualize geometric algorithms with *GeoSheet*, a geometric algorithm is divided into two concurrent but coordinated subtasks. One subtask is a conventional procedural program that concentrates on the aspect of computation, and is referred to simply as *program* in this article. The other takes charge of detailed graphical display operations and is referred to as *visualization subsystem*. To visualize an algorithm, the program uses a set of object-oriented geometric classes to interact with the visualization subsystem. These classes are called *visualization classes*. Visualization classes give the specifications of generic *GeoSheet* visualization operations and can be derived from existing non-visualization algorithm and data structure classes such as LEDA[23,25] in a straightforward manner by following object-oriented approach. When created, the instances of visualization classes are provided with

[a]Refer to the World Wide Web pages at http://www.eecs.nwu.edu/~theory/geomamos.html for more detailed description of this project.

inherited operations and states for both algorithmic and visualization support. The program can use member functions of the visualization base class for graphical operations, or just invoke member functions of the algorithmic base class for ordinary algorithm execution. The visualization requests from the program are passed to the visualization subsystem through a *remote procedure call* (RPC) for graphical data communication and synchronization. Through RPC, the current state of visualized objects is sent, and copied to GeoSheet[b] for display. The user can then use direct manipulation operations to view and graphically input the data required for algorithm execution.

As the display is inherited from Xfig, GeoSheet supports a wide spectrum of graphical objects manipulation mechanism. Both the input and output data can be visualized, printed or exported for other purposes. Its direct visualization manipulation style is useful for algorithm designers for testing their ideas in the development stage, and helpful for debugging the code during the implementation stage. It is also useful for practitioners who would like to visualize how an algorithm performs on various input data after it is implemented.

Having eliminated the need to handle actual visualization operations, such as when to process interactive mouse messages or how to draw on the canvas using *z-buffer* or display context of X windows, the program need not maintain complex visualization specification and runtime structures. Therefore the programmers are free from the task of tedious graphical manipulations and the time needed for algorithm implementation can be significantly reduced. Furthermore, the message invocation mechanism allows transfer of data and control messages between processes in disjoint memory space (or name space) and is therefore capable of supporting computation and visualization activities among machines in a computer network. Since all process communication and synchronization can be done via message invocation, the visualization subsystem and the program are not necessarily restricted to be in the same program name space and can therefore be implemented with different programming languages, or on heterogeneous machine architectures. The program only needs to issue invocation messages and need not concern about how messages are translated into actual input/output operations on graphical display devices. By this mechanism *GeoSheet* achieves its independence of the display device.

Another major advantage is the use of object-oriented visualization classes which can improve reusability. We derive `GeoObject` classes from reliable existing geometric classes by adding a set of RPC style *GeoSheet* invocation operations. The usage of original geometric classes will be the same and all the inherited, available data structures and algorithmic operations can be reused. Therefore to reuse existing programs for visualization, the user only needs to modify geometric class declarations and replace the original Input/Output statements with graphical Input/Output operations. The remaining parts of the programs, including program control flow and geometric member function invocations, will remain the same as the non-visualized version. Examples illustrating this point will be given later.

[b] We use *GeoSheet* to refer to the tool, and use GeoSheet to the visualization subsystem that contains an actual display sheet.

Compared to window based visualization programming schemes, this approach can reduce program development effort for most algorithm developers. For geometric class developers, the implementation of graphical Input/Output interface routines is also straightforward since the sequences of *GeoSheet* invocation messages are fixed and regular. The parameter marshaling operations are in regular order and are almost identical for all geometric objects. The only difference among various kinds of geometric objects is the retrieval of the contents of these objects. Visualization classes can also effectively hide the complex RPC invocation style from users and spare visualization implementation efforts. Our experience has indicated that modification of LEDA's graph algorithms for visualization is quite straightforward and can be finished within days by deriving GeoGraph visualization class from LEDA graph class.

In the remainder of this article, we first examine some of the previous work in this area and review major geometric algorithm visualization concepts in Section 2. The algorithm visualization interface and methodology are described in great details in Section 3. Section 4 discusses the usage of *GeoSheet* for algorithm visualization, followed by two examples given in Section 4.4. Section 5 explores the design of our visualization subsystem. Our distributed support mechanism, geoIPC, is discussed in Section 6. The implementation details and its status are presented in Section 7, and the conclusion and future work is given in Section 8.

2. Related Work

There are several existing visualization systems that were aimed to help geometric algorithm design. These systems provide the visualization support at various stages in the geometric algorithm design (abbreviated as GAD) life cycle, and therefore have different features. Unification of these systems is difficult since the Input/Output interface, file formats, and graphical display environment are different. Systems, such as "XYZ GeoBench" and "Workbench" do consider most GAD phases, but provide primitive visualization support.[20] We briefly describe some of these systems below:

1. Balsa.
   Brown and Sedgewick,[13,14] designers of Balsa (for Brown University Algorithm Simulator and Animator), were pioneers in using algorithm animation to teach algorithm design. Balsa and **Zeus**, which was implemented by Brown,[9] capture meaningful state changes (significant events) in the program by adding procedure calls to the program in a process called *program annotation* for visualization purpose. It emphasizes demonstration of algorithm execution.

2. Xtango.[30]
   This is a system developed by Stasko in Georgia Institute of Technology for algorithm animation by using the path-transition paradigm to add animation to program interfaces.[30] It attempts to simplify program visualization focused on minimizing the effort required to construct display code. It is implemented in C and based on four abstract data types: location, image, path, and tran-

sition, with which the user constructs graphical objects and defines display events. Producing animation sequences in this paradigm involves creation and modification of instances of the data types through program annotation. The system maps program annotations to display events using a finite-state transducer. This design has precise semantics and specifications for the data type, resulting in a rigorous framework for describing the actions that occur in a 2-D animation under an X-window environment.

3. LEDA.[23,25]
   This is developed by a group led by Mehlhorn at Max Planck Institute für Informatik, Germany. It provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. In the LEDA-N-3.0 version (non-template version),[c] it includes most of the data types and algorithms described in the textbooks on algorithms. LEDA gives a precise and readable specification for each of the data types and algorithms mentioned above. It is implemented in an object-oriented style with a C++ class library and can be used free of charge for research and teaching. Its emphasis lies in data structure/algorithm reuse and demonstration.

4. GeoLab.[15]
   It is developed as a programming environment for implementation, testing and animation of geometric algorithms. This tool uses shared libraries of algorithms and an incremental approach to aggregating new types of geometric objects, data structures and extensions accessed through dynamic linking. It runs on SparcStations under Sun/OS using XView graphics library. It is written in C++ and has been used as a research and teaching tool at the Universidade Estadual de Campinas.

5. GeomView.[24]
   Developed at the Geometry Center, GeomView is an interface for viewing and manipulating geometric objects. It can be used as a stand-alone *viewer* for static objects or as a *display engine* for dynamic objects produced by other programs. Its design is based on observations that many aspects of the display and interactive parts of geometric software are independent of geometric content and can be collected together in a single piece of software that can be used in a wide variety of situations. The users can concentrate on implementing the desired algorithms and leave the display aspects like 3-D displaying, colors, rotating, shading, moving, etc., to GeomView. It runs on Silicon Graphics(SGI) IRIS workstations and NeXT workstations.

6. XYZ GeoBench.[27]
   It is designed by Nievergelt and Schorn[27] as a unified geometric programming environment, providing tools for creating, editing, and manipulating geometric objects; demonstrating and animating geometric algorithms; and reducing difficulties in the implementation, application and evaluation of geometric

[c] For updated version, see http://www.mpi-sb.mpg.de/LEDA/leda.html.

algorithms. It provides a user interface as well as a library and is implemented in an object-oriented style running on Macintosh.

7. WorkBench.[18]
   This is similar to "XYZ GeoBench". It emphasizes comparisons/analysis of the quality and usefulness of algorithms (in terms not only of worst-case complexity, but of execution time and space, ease of implementation, robustness, handling of special cases, and average case performance), whereas XYZ GeoBench focuses on robust implementation of fundamental algorithms. It is written in Smalltalk/V in an object-oriented style running on Macintosh.

8. GASP.[31]
   This is a tool developed at Princeton University for geometric algorithm animation. It has a library of primitive animation functions. The user stores the needed animation routines in a *style* file, and in the main application program the user will invoke these animation routines at appropriate places. All input and output operations are performed using files. The software written in C is available at ftp.cs.princeton.edu:/pub/people/ayt/gasp.tar.Z via anonymous **ftp**.

9. Mocha.[4,5,6]
   This is a distributed model with a client-server architecture for algorithm visualization over the Internet. In the Mocha model, only the interface code is exported to the user's machine, while the algorithm is executed on a server that runs on the provider's machine. Mocha provides high levels of security, protects the algorithm code, places a light communication load on the Internet, and allows users with limited computing resources to access executions of computationally expensive algorithms. Prototypes of web-based geometric algorithm visualization systems based on the Mocha model have been developed at Brown University and at the Max-Planck-Institut für Informatik[d] Mocha is part of the broader vision of GeomNet,[7] a system for distributed geometric computing over the Internet.

10. Others.
    Ten different geometric display systems were included in a videotape edited by Brown and Hershberger[11] and distributed at the 8th Annual Symposium on Computational Geometry. At the 9th Annual Symposium on Computational Geometry another videotape containing 8 segments was made available.[12] Each has demonstrated the power of animation in conveying the key ingredients of the algorithms being implemented. We mention here two more visualization tools, **Alpha-shape**,[16] developed at the National Center for Supercomputing Applications, University of Illinois, is a special purpose *shape* modeler, and **Shastra**[2] developed at Purdue University produces several in-

[d] Refer to `http://www.mpi-sb.mpg.de/LEDA` for a link to LEDA-in-the-web, and `http://www.cs.brown.edu/cgc/demos/Mocha.html`.

teroperable, collaboration-aware toolkits for the creation, manipulation and reconstruction of solid models from a variety of sources.

3. Geometric Visualization Classes

Visualization classes are used as a uniform interface to all geometric class objects for programs to invoke visualization functions. Objects of visualization classes provide algorithm developers with data structures and computation functions as well as interfaces to GeoSheet that allow direct manipulation style graphical input/output operations. In this section, we will describe an object-oriented scheme that allows the user to derive visualization classes from existing geometric classes. This scheme also provides two generic graphical input/output functions whose usage follows traditional input/output paradigm in user's program.

3.1. Basic Visualization Class Concept

A *geometric class* pertains to a specific geometric object along with a set of data structures and functions which can be used in a program. Examples which are widely supported in many existing geometric classes include point class, line class and polygon class. A *visualization class* is derived from a geometric class and supports invocation of *GeoSheet* visualization functions of the geometric object specified in the geometric base class. Specifically visualization class is constructed by multiple derivation from a geometric class and a visualization generic class, `GeoObject`; and hence inherits both geometric and visualization operations of these two base classes. For example, suppose a visualization class `GeoPolygon` is derived from `Polygon` and `GeoObject`. The resulting `GeoPolygon` will support both geometric functions such as determining whether a point is inside a polygon, and visualization functions such as drawing a polygon on GeoSheet.

According to our experience, a good visualization class should meet the following requirements:

1. *Data abstraction.* Visualization class should hide algorithm-unrelated details from its users. Algorithm developers should be able to generate required visualization effects without using any mechanical but tedious operations such as graphical display functions, GeoSheet binding, error handling and data communication of visualization objects between GeoSheets. This requirement is difficult to fulfill if programs are developed using X11, GL or MS Windows.

2. *Reuse of existing geometric classes.* For library providers, one of our important design consideration is to provide a methodology that allows users to reuse available geometric classes such as LEDA, instead of re-writing the whole visualization classes from scratch. This requirement is important for practical applications when we want to develop a visualization system that provides *closed* visualization libraries. Although many geometric classes are similar in their built-in functions, it is not trivial, however, to meet such a requirement

since their interfaces differ greatly. That makes reuse of existing classes almost ad-hoc, and limited only to specific libraries.

3. *Easy to be integrated to current programming environment.* Algorithm development activities involves not only programming but also testing, verification and debugging. These activities heavily rely on assistance of programming development tools such as interactive debugger. Visualization classes should be ready to be integrated smoothly into available program development methodology and tools. For instance, it will be useful if one can utilize current debugger to debug visualized programs. This issue has not been addressed in previous work.

To support reuse our approach uses object-oriented programming paradigm (in C++) in visualization class design. By using *data encapsulation* mechanism, the detailed *GeoSheet* invocation interfaces and data transfer operations are encapsulated in visualization objects and are hidden from programmers. A visualization class inherits any C++ geometric class and reuses the operations and implementations of the geometric base class. By providing a mechanism to *override* the visualization specification in `GeoObject`, the derived visualization class can be used in supporting geometric computation as well as visualization with a unified class interface. The resulting visualization objects retain from inheritance the same interfaces of geometric functions and do not change the control or data flow of original programs.

3.2. `GeoObject` - *Geometric Generic Class*

`GeoObject` class is required to be one generic base class of all visualization classes in order to conform to *GeoSheet* invocation style. The definition of `GeoObject` is as follows:

```
class GeoObject {
protected:
   FLAG Persistence;      // Persistent object indicator
public:
   GeoObject()    { /* Clear persistence. */};
   virtual Graphic_Read()  {};
   virtual Graphic_Write() {};
   void SetPersistent()   { Persistence = ON;};
   void ClearPersistent()   { Persistence = OFF;};
   };
```

The `GeoObject` class defines a common visualization interface to visualization subsystem for all visualization classes, `Graphic_Read` and `Graphic_Write`, for which a variety of derived visualization classes can provide different implementations for their own structures. `GeoObject` can also be used as a mechanism to check if every visualization class is conformed to our visualization style and reduce possible errors that may occur later. One major advantage of using `GeoObject` is that it provides a clear, unambiguous visualization specification. This well defined visualization specification is also important in incremental software development. Since both usages and internal structures vary greatly from one geometric class to another, it is not trivial to define a consistent interface to all visualization classes. With this common specification of `GeoObject`, visualization class designer can easily understand the visualization class requirement and concentrate on the implementation without spending efforts on designing his/her own visualization interface. At the same time, the user can utilize all available visualization objects through a uniform high level graphical interface. This certainly can simplify and speedup the program development process. Another advantage is that `GeoObject` base class can be used as common runtime base to all visualization objects. This feature is useful in implementing dynamic runtime functions whose real class type will not be known until the program actually executes. For example, to draw all the objects in a linked list containing heterogeneous visualization objects requires the type of each object in order to invoke correct drawing routines for the objects. To be more specific, you need to use polygon drawing function to display a polygon and use point drawing function for a point. With the virtual function mechanism in `GeoObject`, the drawing function that matches the class of each object will be correctly invoked. This feature that all derived visualization classes use the same member functions is known as *polymorphism*.[17] The `GeoObject` data members are used for visualization object management, and will be discussed in Section 3.6.

We remark here that we can also construct visualization classes directly from geometric classes without inheriting the `GeoObject` base class. This approach is quite straightforward and allows the user to define more visualization functions for complex geometric objects and fewer visualization functions for simpler ones. However, it does not provide the common runtime base as described earlier. We adopt the former approach because we believe a consistent and conformable methodology is critical to the generality of visualization classes.

3.3. Visualization Class Derivation

We use C++ multiple inheritance mechanism to derive a visualization class from a geometric class and `GeoObject`. Such derivation is straightforward and is the same as ordinary C++ derivation. For example, suppose that we want to create a new derived visualization class `GeoPolygon` from an existing geometric class `Polygon` and `GeoObject`. The specification of `GeoPolygon` is as follows:

```
class GeoPolygon : public Polygon, public GeoObject { // Derivation
    GeoPolygon (pointlist& plist) : Polygon(plist) ;
    ~GeoPolygon ();
    Graphic_Read(int GeoSheetID);
    Graphic_Write(int GeoSheetID);
};
// The implementations of constructor, destructor, Graphic_Read
// and Graphic_Write follow.. . .
```

We will describe the general rules for implementing these functions in later sections. Nevertheless, with this derivation the specification of this new visualization class, which extends polygon class to visualized polygon class, is completed. The `GeoPolygon` user can simply replace the declaration of `Polygon` class in the program with `GeoPolygon` and is ready to visualize the polygon in the original program. For example, an object `P` of the class `Polygon` that was created by:

```
Polygon P;
```

can now be replaced with:

```
// Polygon  P; replace Polygon by GeoPolygon
GeoPolygon  P;
```

The visualization object `P` is created by using C++ object creation operations and can be initialized when created or set by subsequent `Graphic_Read` operations.

3.4. Visualization Object Invocation

Visualization objects can be invoked with two groups of operations; one is the original geometric algorithmic functions inherited from the geometric base class, and the other is visualization functions defined in `GeoObject` generic class. The user may use visualization operations in the program to visualize the current state of an object on a GeoSheet. We will describe several techniques using visualization objects to visualize geometric algorithms in Section 4. Here we use an example to illustrate the basic visualization object usages. Suppose the user wants to draw an arbitrary polygon on a GeoSheet, and to repeatedly input query points to test the `Inside` function of a polygon class. At the same time the user also wants to test the `AddVertex` function as follows. If a point is inside the polygon, it will be added to the polygon and the new polygon should be shown on GeoSheet. The user can easily verify these polygon functions by comparing the output messages, the polygon and the input point position on the GeoSheet as follows:

```
main()
{
  GeoPolygon P; // P is a visualization object.
  P.Graphic_Read(); // Ask default GeoSheet to input a polygon.
  while(i < N)
  {  // A query loop.
     GeoPoint Pt;  // Pt is visualization object derived from Point.
     Pt.Graphic_Read(); // Input a query point from default GeoSheet.
     if (P.Inside((Point)Pt)) == TRUE)// Note the base conversion of Pt.
     {
         // Display the message the query point is inside the polygon P.
         // ...
         // Add this point to polygon and display it.
         P.AddVertex((Point)Pt);
```

```
            P.Graphic_Write();  //  Display new polygon.
        }
        else
            // Show the query point is outside the polygon.
    };
};
```

In this example visualization object `P` is input graphically with `P.Graphic_Read` and displayed with `P.Graphic_Write`. `P` can also invoke the `Inside` and `AddVertex` functions to compute the geometric characteristic of polygon. Notice that both `Inside` and `AddVertex` functions expect an ordinary point object, rather than a visualized point object. We need to use `(Point)Pt` to adjust the base part of the visualization object to its geometric base part. It is important to note that the program structure including data dependence and control flow has not been modified by the usage of `Graphic_Read` and `Graphic_Write`. This feature is very similar to the usage of C `scanf` and `printf` statements. Not only does the feature make visualization task natural to use, it also makes debugging easy. We can make use of existing debugger to debug visualized program and can display ordinary alphanumeric data and objects in graphical form. For example, we can invoke `P.Graphic_Write` to display the selected object `P` on a GeoSheet and invoke traditional `print` facilities of debuggers to print its alphanumeric contents.

3.5. Implementation of Visualization Functions

We now describe the implementation of `Graphic_Read` and `Graphic_Write` operations of each visualization class. As described in the previous section, these two operations are actually interface routines to GeoSheet, rather than the actual graphical display routines for visualization objects. They do not involve any graphical operations. Since GeoSheet and the program are two disjoint, independent processes and cannot directly share data with each other, it incurs an overhead of passing control and data between the two subsystems. For the program to successfully pass the visualization requests to the destination GeoSheet and vice versa, the interface is required to carry the correct information across name space boundary.

We use a *remote procedure call* (RPC) scheme to solve this problem. Since GeoSheet is a message driven process communicating with the source that initiates the request, it is natural to regard each graphical read or write request as a read or write RPC function. Besides specifying read or write operations, the RPC should also contain the object type and other possible arguments required when switching control to the read/write routines of the invoked object. GeoSheet then uses this information to start the corresponding functions for graphically reading or writing an object requested by the program. Figure 1 presents the control flow of the general graphical read and write operations.

We note that the only argument required of the program for graphical read and write is the GeoSheet index. Therefore the program fragments for graphical read and graphical write of each visualization class are as follows. Let `GeoVisObj` be a

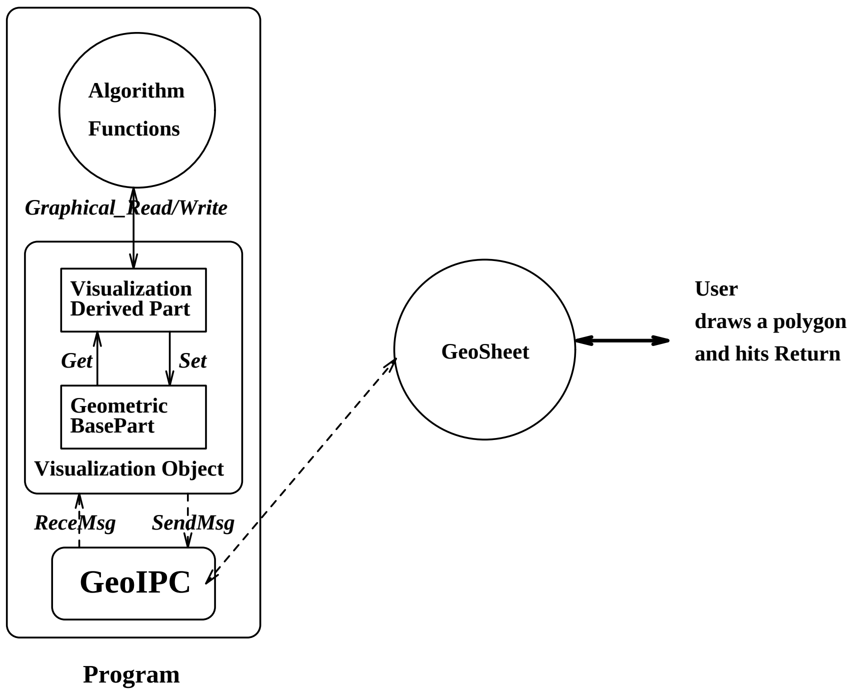

Fig. 1. Control flow of graphical read and write operations in visualization objects

visualization class. The pseudo-code for `GeoVisObj::Graphic_Read` is:

```
(1).  Use GeoSheet index to locate the GeoSheet port.
      // The GeoSheet IPC mechanism will be described later.
(2).  Allocate a message buffer.
(3).  Fill the message buffer with the read request and the
      visualization class type.
      // Note that class type is finite in length for geometric class
      // primitives.
(4).  Send the request message to GeoSheet port.
      // GeoIPC guarantees that the message be reliably sent to target
      // GeoSheet.
(5).  Suspend and wait for the reply message, which is to hold
      the input data.
(6).  Resume once the reply message arrives.
(7).  Extract from the reply message the desired data.
(8).  Assign the data to the geometric base part of the visualization
      object.
(9).  Return to calling program.
```

and the pseudo-code for `Graphic_Write` is:

```
(1).  Use GeoSheet index to locate the GeoSheet port.
      // The GeoSheet IPC mechanism will be described later.
(2).  Allocate a message buffer.
(3).  Fill the message buffer with the contents of the geometric base
      part of the visualization object.
(4).  Send the request message to GeoSheet port.
      // GeoSheet will display the object according to the message
      // contents.
(5).  Suspend and wait for the acknowledge message.
(6).  Resume when acknowledge message arrives.
(7).  Return to calling program.
```

We point out two important elements in these two procedures. The first one is *access* of geometric base part of visualization objects. Note that only the most fundamental access operations i.e., read and write are required. The other is the *geometric message protocol*, which must be followed when sending or receiving messages to and from GeoSheet so that correct data can be sent between the program and GeoSheet.

Compared to the X11 based visualization protocol, the geometric message protocol is very simple as it only concerns the components of geometric objects such as object type and the data, whereas the X11 messages are more complex and less efficient. However, from our experiences and the usage of widely used graphical editors, such as Mac Draw and Xfig, this geometric message protocol is sufficient for input and output of geometric objects. It is reasonable to require that all visualization class providers follow this protocol to define their visualization services. One can also derive complex visualization classes from these visualization primitives based on this protocol. For instance, one can use the segment and point primitives to draw a polyline.

3.6. Consistency of Visualization Class Object

A potential problem that is caused by repeated reads and writes on different GeoSheets of a visualization object is that of what is referred to as *visualization object consistency problem* and must be addressed. At the present time, a visualization object is created as an ordinary C++ object in the program, and then it can be 1) repeatedly written to one GeoSheet, 2) repeatedly written to different GeoSheets, 3) repeatedly read from one or different GeoSheets, or 4) combination of these previous cases. Any of the above operations will result in more than one copy of the same graphical object among active GeoSheets. Therefore if any of these copies (including the visualization object in the program) is updated, all the other copies will become *stale*; that is, their contents are no longer consistent with each other. The problem will be more complicated if we allow GeoSheet users to update objects directly on the GeoSheet. In addition, if a visualization object is *deleted* in the program; then all the copies representing this object have to be located and purged in all active GeoSheets.

In the current release we assume programmers only use GeoSheet as a graphical input/output worksheet and users will handle the object consistency problems themselves. We propose a validation protocol by using an `ActiveGeoSheet` object and a visualization object index scheme in `GeoObject` to solve the object consistency problem. An object index must be unique within one GeoSheet and can be used to locate a specific visualization object in a particular GeoSheet. `ActiveGeoSheet` of a visualization object is designed to contain all object indexes of the visualization object in different GeoSheets and provides the following functions:

```
int status = ActiveGeoSheet.Put(int GeoSheetID, ObjectIndex iObject,
                                GeoObject *pObject);
ObjectIndex iObject = ActiveGeoSheet.Query(int GeoSheetID,
                                           GeoObject *pObject);
int GeoSheetID = ActiveGeoSheet.GetNext(GeoObject *pObject);
```

Method `Put` will store the tuple, `GeoSheetID`, object index and `GeoObject` pointer to the `ActiveGeoSheet` object. `Query` is used to locate an object in a particular GeoSheet. It will return the object index if it was put before; otherwise a null value will be returned. `GetNext` is used to find all GeoSheets that contain a given object. Whenever a visualization object is created on a GeoSheet, the tuple consisting of `GeoSheetID`, object index and `GeoObject` pointer is stored into `ActiveGeoSheet`; similarly, the tuple is removed when the object is discarded. Therefore from `ActiveGeoSheet` we can have all copies of the object in various GeoSheets.

We are now ready to describe the protocol. If a visualization object requests a graphical read operation from any GeoSheet, it is obvious that all other copies of this object, if present in other GeoSheet will be "stale" after the read operation. Hence we can simply traverse the `ActiveGeoSheet` and *update* all the object tuples to maintain consistency.

The graphical write operation is more complicated because an object will become "dirty" by either a read or a program update. A write operation will by no means make an object "stale". As a result, if an object is not written to GeoSheet for the first time; then it has no way to tell whether this object is dirty or not since its last read or write operation. To solve this, we adopt a *pessimistic* invalidation protocol. That is, whenever an object issues a second `Graphic_Write` operation, we invalidate all the existing objects in order to ensure object consistency. This scheme is pessimistic since it introduces overheads of unnecessary update operations for "clean" objects. However, we believe such overheads will not be significant enough to degrade the system performance because most write operations are one shot according to our experience. Only objects involved in geometric queries and animation will need to be repeatedly written to GeoSheets.

We remark that under some circumstances there might exist inconsistent objects in our current scheme when visualization objects are updated by the program. There will be a period during which the objects are inconsistent until a read or write operation is issued. Unless we use a separate monitoring process to watch any change of a C++ object, it is very difficult to detect inconsistency caused by the program. We believe this inconsistency is unavoidable under the current C++ scheme. Nevertheless, the period of inconsistency can be reduced by immediately writing the updated visualization object to a GeoSheet.

3.7. Persistent Visualization Objects

Another problem comes from the requirement of *persistent objects*. Under the current scheme, the life time of visualized objects is the same as ordinary C++ objects. All visualization objects are automatically purged by C++ run time mechanism after the program terminates. However, we may require for some applications that visualization objects remain available on GeoSheets after the program terminates. For example, program `A` may need to use the output of program `B` so the output of program `B` should not be purged.

Persistent objects can be supported by several ways. One scheme is that the user interrupts program execution, uses the filing mechanism of *GeoSheet* to store the current contents of GeoSheets into files before the program terminates and reload the data file when other program needs it.

Another more general scheme is to use a private variable, `persistence`, in the `GeoObject` part to indicate the persistence of the visualization object. Two operations, `SetPersistent` and `ClearPersistent`, are provided for the persistence control. If the `persistence` is set, then the destructor of the deleted visualization object will not send the purge request to GeoSheets where the visualized copies reside after the program terminates. The next program may clear or keep the `persistence` of visualization objects after reading these persistent objects. This persistent scheme provides better granularity of visualization objects and supplies more flexibilities in supporting object sharing.

4. Algorithm Visualization using *GeoSheet*

Visualization of geometric algorithms requires the capability of entering as input geometric objects and viewing as output the computation in visualized form. In this section, we will describe our geometric algorithm visualization paradigm based on the notions of logical work sheets and visualization object. The visualization procedures in our model include allocating a set of logical work sheets and invoking visualization operations by using the logical work sheets as target display devices. The invocations will then be translated to actual graphical read or write operations on the destination GeoSheet. Our visualization paradigm is similar to that of `scanf` and `printf` routines in C programs. Algorithm developers need not know about technical details such as the I/O device type or display control code in order to visualize geometric objects.

4.1. Logical Work Sheet

A logical work sheet is a C++ object in the program representing a GeoSheet port connected to the visualization subsystem. Each logical work sheet is viewed as an independent source of input or output or both for `Graphic_Read` or `Graphic_Write` visualization operations. The visualization requests from the program are sent to GeoSheet through the attached logical work sheet. Each visualization invocation must specify a logical work sheet as the target input/output device; otherwise a default work sheet (`stdin` and `stdout`) will be used. The relationship of the logical work sheets to programs is similar to the files to processes.

To support high level, user-friendly visualization of geometric algorithms several features of the logical work sheet are considered:

- *Supporting multiple work sheets in one program execution.* For algorithm development or testing, it is useful to have several Input/Output devices to multiplex the computation results at different stages of the execution. For example, a work sheet can be used as `stdin` of a program, another as `stdout`, the third as an intermediate work sheet and the fourth can be one for displaying results to be exported to other programs. See Section 4.4 for more details. Our current implementation supports up to 15 GeoSheets.

- *Supporting static and dynamic binding scheme.* The binding of a logical work sheet and a GeoSheet can be determined statically or dynamically. In the static scheme when a program allocates a logical work sheet, a new GeoSheet will be created and attached to the program. This GeoSheet will terminate itself at the end of the program execution. In the dynamic scheme the creation and termination of GeoSheets are independent of program execution. When a logical work sheet is created, the GeoSheet service library will prompt the user for a binding. Such a binding currently asks the user to specify a set of IP information to locate the target GeoSheets. A tool, *GeoPanel*, is designed to support such specifications. Only the connection, rather than creation is performed in the dynamic scheme. For individual programs, the static scheme is sufficient since it is not required to export computation results to other programs. On the other hand, the dynamic binding scheme is useful in situations where computation results are shared by different programs or comparisons of different algorithms are needed. Figure 2 shows an example of two programs sharing a GeoSheet by using the dynamic binding feature.

- *Display device independence.* One main design purpose of logical work sheet is to provide graphical device independence. When logical work sheets are created, no specific display information is required. Only the binding information is needed to establish the connection to a GeoSheet. Logical work sheets only serve as communication channels for the visualization operations and results, and do not themselves implement any functions dependent on display characteristics.

- *Supporting distributed visualization.* One key feature to support algorithm visualization in a distributed environment similar to Refs. [2,3] is to provide a mechanism that multiple programs can view or receive graphical data from a set of distributed but shared GeoSheets. Our approach is to use a logical name to refer to a distinct GeoSheet under the control task of *GeoPanel*. All the algorithmic processes can ask *GeoPanel* to use this name to locate the associated GeoSheet for graphical Input/Output operations. The results displayed on the GeoSheet associated with this common name can therefore be shared. Please refer to Section 6 for more detailed descriptions.

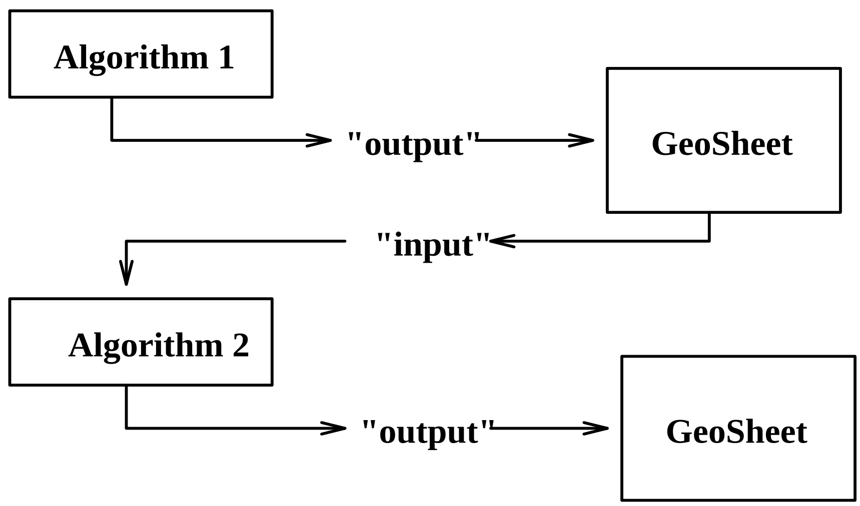

Fig. 2. A mapping of two algorithms sharing a GeoSheet

To sum up, the logical work sheet scheme can provide flexible GeoSheet usages. Different programs can use an existing work sheet as a platform for communication. Similarly, a program can use different GeoSheets to reflect different stages of computation. Being device independent, the GeoSheet, which in its present form is a 2D display device, can actually be a 3D display, and it can be implemented on different machines, as long as it supports all the visualization operations and data types described in Section 5.3. Besides, the separation of runtime structures of visualization subsystems and program also can simplify the implementation of these two subsystems. The reason is that the program is free from direct manipulation of these visualization data structures and the data flow and data segment size will become less complicated. Similarly, the visualization subsystem only needs to deal with a fixed set of message requests and return the results via the message interface, and hence can be independently developed and refined in parallel to algorithm development. Therefore algorithm implementation will not be affected by rapid changes of display technology. The separation also lets GeoSheet and the program work in client and server execution model, and is helpful in developing distributed collaborative visualization.[2] We will describe this aspect in more details in Section 6.

4.2. GeoSheet Service Setup

Before a program starts visualization operations, the program needs to perform a GeoSheet service setup procedure to create logical work sheet objects in order to establish the connection between the program and GeoSheet:

```
GeoSheetServiceSetup();
```

After the GeoSheet service setup, two logical work sheet objects, *GeoSheetStdIn* and *GeoSheeStdOut*, will be created. *GeoSheetStdIn* connects to a default input GeoSheet, which is used for subsequent `Graphic_Read` operations; and similarly, *GeoSheetStdOut* connects to a default output GeoSheet, which is for `Graphic_Write` operations.

If additional work sheets are required, the program may use:

```
int sheet = NewGeoSheet("string");
```

to allocate a new logical work sheet. An index, *sheet*, will be returned to the program and can be used in subsequent `Graphic_Read`(string) or `Graphic_Write`(string) operations in order to redirect graphical Input/Output operations to the new work sheet. The name *string* can be used as a global name to all programs under the control of *GeoPanel*. To locate the GeoSheet represented by this name, the user may use

```
int sheet = QuerySheet("string");
```

and the index of the logical work sheet will be returned, if available, or an error, otherwise.

4.3. Graphical Input/Output

The input and output operations on GeoSheet are performed via invocations of the visualization objects. If the program originally does not use visualization classes, we use visualization objects as a high level visualization template and copy the desired display data to visualization objects for graphical Input/Output operations. We also provide a set of built-in visualization classes for such kinds of applications. Currently we support a set of visualization primitives which are listed in Table 1. The choice of this set of primitive visualization objects is based on our experience of geometric algorithms, and the set can always be modified.

GeoSheet also supports implementation of geometric queries. In geometric query mode the user is required to input some geometric object interactively during program execution, and then the program can continue the computation such as in ray shooting operation. Geometric query operations may be one shot or repeated. Frequent geometric queries include searching some data objects satisfying certain constraints or properties.

It is straightforward to support the geometric queries using visualization objects. If the query input objects do not belong to the original input data objects, then we can use graphical read operation to input a temporary visualization object as the query input data. However, if the query objects belong to the original input data objects, then the user has to use object selection mechanism to select the object in the input set and pass the selected object to the program for query computation.

Table 1. A set of primitive visualization objects.

Class	operations
`Simple object`	`polygon`
	`polyline`
	`circle`
	`segment`
	`parabola (output only)`
	`point`
	`triangle`
	`rectangle`
	`text`
`Composite object`	`undirected graph`
	`directed graph`
	`weighted undirected graph`
	`weighted directed graph`

We will discuss the object selection in Section 4.5.

In addition to visualization objects, we support two message interfaces for non-graphical queries:

```
void GeoPause(sheet, "string");
int GeoScanf(sheet, "string", format, VarAddr);
```

`GeoPause` will display a confirmation box with the message *string* on the GeoSheet with index *sheet*. Similarly, `GeoScanf` will display a dialog box with the message *string* and an editable line on GeoSheet with index *sheet*. After the user inputs the ASCII string, the `GeoScanf` will scan the input line using the input *format* and store the result to *VarAddr*. For example,

```
GeoPause("Please change the line width to 10 now.");
```

will cause the program to pause, displaying a message box with the above message until the user presses OK button on the `GeoSheetStdOut` to resume execution. Similarly,

```
GeoScanf("Please input the max number of points",
"%d", (char *)&n);
```

will cause the program to pause, displaying a `scanf` box containing the above message on the `GeoSheetStdIn` till the user types a value to resume execution.

4.4. Example Programs

We present two examples here to illustrate the visualization procedures using GeoSheet. The first example concerns the "*art gallery problem* in a simple polygonal region", which is to find a placement of a minimum number of guards, called *vertex guards*, at the vertices of the polygon so that every point of the region is *visible* from at least one guard. It has been shown that this optimization problem is NP-hard.[22] It is known that $\lfloor n/3 \rfloor$ vertex guards are always sufficient, and sometimes necessary,[26] in order to see the entire polygon with n vertices. Because $\lfloor n/3 \rfloor$ guards are only sometimes necessary, it is often the case that the number of guards can be reduced to less than $\lfloor n/3 \rfloor$. In this example, a heuristic method based on the placement of guards that favors reflex vertices over convex ones was implemented. The user is required to interactively select as few reflex vertices as possible for placing guards to see the entire region.

```
(1).   GeoPolygon     P; // P is the input polygon.
(2).   GeoPolygon     R; // R is the visibility region.
(3).   PolygonNode    v;
(4).   GeoPoint       GuardCandidate;
(5).   list<GeoPoint> GuardList;

       // Create logical work sheets for visualization.
(V.1). GeoSheetServiceSetup();
(V.2). int TempSheet = NewGeoSheet("Reflex vertex visibility region");

(V.3). P.Graphic_Read();
(V.4). P.Graphic_Write(TempSheet);

       // To display all reflex vertices before traversing polygon.
(V.5). ShowAllReflexVertices(P);

(6).   forall_node(v, P) {
(7).     if (P.ReflexVertex(v)) { // A reflex vertex is found
(8).       GeoPoint GuardCandidate(v.GetX(), v.GetY());
(V.6).     GuardCandidate.Graphic_Write(TempSheet);
(9).       GeoPolygon NewVisRegion (P.VISIBILITY(GuardCandidate));

           // Display the visibility region of the prospective guard.
(V.7).     NewVisRegion.Graphic_Write(TempSheet);

(10).      if (R.Contains(NewVisRegion) == FALSE) {
             // New visibility region covers some region not in the
             // current visibility region.

             // A visualization function to assist user to decide.
(V.8).       ShowAdjacentReflexVisRegion(TempSheet, P, v);

(11).        char cAnswer;
(12).        printf("Would you like to pick this guard?");
(13).        scanf("%c", &cAnswer);
```

```
(14).        if (cAnswer == 'Y' || cAnswer == 'y') {
(15).           GuardList.Insert(GuardCandidate);
(16).           R.UnionRegion(NewVisRegion);
(V.9).          R.Graphic_Write();
(17).        };
(18).      }        // End of if (R.Contains(NewVisRegion) ...
(19).    }    // End of if (P.ReflexVertex(v))
(20).    if (P.Contains(R) && R.Contains(P)) // Visibility region found.
(21).    break;
(22).  }  // End of forall_node (v, P)

        // Display all the guards and the visibility region.
(V.10). GeoPoint n;
(V.11). forall(n, GuardList) n.Graphic_Write();
(V.12). R.Graphic_Write();

(23).  } // End of main();
```

In the above example we make use of several visualization techniques provided by *GeoSheet* to visualize this non-visualized algorithm. These visualization operations are at lines that begin with `V` and are explained below.

1. We use two output work sheets for this problem. The `stdout` is used to display the final visibility region and guards placement, and the other sheet is for display of visibility region of each reflex vertex. Operations in lines `V.1` and `V.2` set up the visualization service and allocate an additional work sheet.

2. Visualization class, `GeoPolygon`, is derived from `MyPolygon` (not given here), which defines a polygon with a set of vertices and provides several important geometric operations. The member function *Contains* uses a polygon as input and checks whether the associated polygon object contains the input polygon or not. The member function *ReflexVertex* uses a vertex of this polygon object as input and determines whether the vertex is reflex. The member function *UnionRegion* uses a polygon as input and computes the union of the input polygon and the polygon object. `MyPolygon` class is derived from a graph class. With this class hierarchy, `GeoPolygon` objects inherit characteristics of a graph and of a (geometric) polygon. For example, in lines `V.3` and `V.4`, `P` is viewed as a polygon visualization object and can be input and output with graphical operations; while in line `6`, it is traversed as a graph object. In lines `7, 10, 20`, it is manipulated based on geometric characteristics of

a polygon. This usage illustrates a very important feature of object-oriented programming.

3. Visualization Input/Output operations using GeoSheet are similar to those commonly used alphanumeric Input/Output operations. In this example lines `V.3` and `V.4` display the input polygon on two work sheets, lines `V.10` to V.12 visualize the final result of the given input polygon. The visibility region is also displayed in line `V.9` whenever it is updated.

4. The intermediate program state is used to show the status of the geometric algorithm and is usually abandoned after it is examined. `GuardCandidate` and `NewVisRegion` are two intermediate objects used in the main loop of this program. They represent the current reflex vertex and the visibility region respectively. Visualization operations in line `V.6` and line `V.7` display the current state of these two objects in each iteration. At the end of each iteration, these two objects will be purged by C++ run time management and therefore discarded from the work sheet.

5. The user can supply more high-level visualization utilities based on the fundamental operations of visualization objects. For example, we used the following heuristic to decide if a reflex vertex is to be inserted into the guard list, i.e., a reflex vertex is selected if the newly covered region $(Q \setminus R)$ is not contained in the visibility regions of the subsequent or previous reflex vertex. This is what `ShowAdjacentReflexVisRegion` in line `V.8` is used for.

Note that most visualization statements can be removed without affecting the original control flow of the program. The fact that human interaction can be incorporated at the source code level to control the output of an algorithm enables the user to fine-tune a heuristic when developing approximation algorithms for NP-hard problems.

The second example shows a user-defined geometric class facilitated by the GeoObject class.

(1) It is straightforward to build a new visualization class composed of primitive visualization object types listed in Table 1, as shown in the following "λ-geometry Steiner Tree" example.

(2) When a user-defined geometric class is to be constructed, a new primitive class should be implemented. This involves implementation of a new conversion routine which is a bridge between *GeoSheet* and the library of geometric visualization classes (the visual annotation for each of the classes), *GeoLIB*. This part is dependent on the display device. We skip its description.

The λ-Steiner tree class is given in the following C++ code fragment. Note that the member function Graphic_Write() can be overloaded according to the parameter types and is implemented by composing visualization object types.

```
class SteinerTree : GSObject {
public:
  geoPointList   P;         // input point set
  geoPointList   Q;         // Steiner Point set
  geoGRAPH       ST;        // Steiner tree
  int            lambda;    // number of orientations
  int            ST_length; // length of the Steiner tree

  SteinerTree(geoPointList P);
  ~SteinerTree();

  Graphic_Read();  // input method is inherited from
                   // geoGRAPH.Graphic_Read().
  Graphic_Write(int topology_flag, int label_flag, int sheetId);
     // IF topology_flag is ON=1, the graphic output will
     //    display the Steiner tree topology by using
     //    geoGRAPH.Graphic_Write().
     // IF topology_flag is OFF=0, the graphic output will
     //    display the actual layout of the Steiner Tree each of
     //    whose edges is in a feasible orientation.
     //    Note: This specific output needs to be
     //    implemented by using segment.Graphic_Write().
     // IF label_flag is ON=1, lambda-distance will be
     //    displayed on each edge as a label,
     //    implemented by setting geoGRAPH.edge_label().

  Graphic_Write(geoPointList Q, int highlight_style, int sheetId);
     // The Steiner point set will be highlighted, implemented
     //  by using geoPointList.Graphic_Write(highlight_style).
}
```

For more information about interactive visualization operations on the λ-gemoetry Steiner tree problem, such as Insert(), Select() which assist the user to adjust the output Steiner tree on the fly, see Ref. [28].

4.5. Geometric Object Selection

One requirement for geometric algorithm visualization is to be able to locate an object from a set of geometric objects displayed on a GeoSheet. For example, in the problem of finding the shortest path tree from a single source vertex to all other vertices of a simple polygon, the program may request the user to *select* one of the vertices as the source, and then begins the computation of the shortest path tree. The requirement presents a fundamental communication problem, as the selection is performed on GeoSheet, but the execution on the selected object is to be carried out by the program, which is a process disjoint from GeoSheet. The solution depends on how one represents the selected object in each process. For example, a point in the program may be represented as an object *point*, a pair of x- and y-coordinates or an index to a point storage. Similarly, GeoSheet may use one of these representations for a point. However, the object pointer in the program is meaningless to GeoSheet and vice versa. We have two options. The first one is to use an ordered pair, <*GeoSheet, ID*>, containing the GeoSheet logical index and object ID, which is stored in the visualization object in the program. However, as we have already seen in Section 3.6, this will require an individual object to store a list of such pairs in order to locate all the copies of visualization objects in GeoSheets. That will incur an overhead for simple objects such as point and lines. Another way is to use the contents of the objects, such as x-, y- coordinates of a point, to match the contents of objects in GeoSheet. The drawback of this scheme is the overhead needed to search for a matched pair, but it is efficient in storage, giving a time-space tradeoff. According to our experience, only simple geometric objects will be used for most applications. Thus the content-matching scheme is more suitable than the index scheme.

The visualization object selection plays a key role especially in implementing heuristics. Due to its implementation difficulty and lack of general usage, the interactive selection of visualized objects is not commonly seen in current interactive tools. Our scheme attempts to address such an issue by combining the interactive message protocol at the GeoSheet and the visualization interface. This scheme puts the burden of performing the matching of the visualization objects on the program side, and reduces the complexity of GeoSheet operations. However it reduces the interactive graphical operations to a sequence of device independent message invocations and conforms to our general visualization class message architecture without putting special restrictions on visualization subsystems.

4.6. Geometric Object Update

Another frequently used operation is to interactively update objects in a given set of objects. For example, the user might want to interactively change the positions of some sites in a Voronoi diagram and see the difference caused by the update. This kind of operations requires complicated communication operations between the program and visualization subsystems.

We introduce a *copy-to-copy-back* scheme to meet such requirement in updating a set of objects. The basic philosophy is to first copy all the objects on the GeoSheet to a newly created list, when an update is requested. Then prompt the user to update any objects and press *return* when update operations are finished. When finished, GeoSheet uses the list to scan the objects in it and updates the changes, copies back the current contents of the list into a message, which is returned to the program.

After the *reply* message arrives at the program, the visualization update routine copies back the contents from the message to the object and resumes the execution. At this point, the contents is updated.

5. GeoSheet Design

This section describes the design of *GeoSheet*. *GeoSheet* is modified from Xfig and offers most frequently used Xfig functions of interactive graphic objects drawing and manipulation. In addition, it extends the capability of Xfig in a number of ways including a message driven interface for geometric algorithm invocation, an object drawing and manipulation tool box containing most fundamental geometric objects used in geometric algorithms, and distributed supporting functions.

5.1. GeoSheet Architecture

GeoSheet consists of user-interface message handling routines, algorithm invocation message conversion routines, X11 drawing action routines and algorithm execution facility. Figure 3 shows the internal block architecture of *GeoSheet*.

- **Main Panel:** The main panel is a graphical user interface to process interactive mouse and keyboard input actions for direct manipulation operations. Four kinds of user interfaces are supported by the main panel. The first kind is the menu for file, print, redraw, delete and file export operations. The second is the geometric visualization object creation/deletion. Tool boxes of this category contain most Xfig object drawing/editing tools. In addition, we also support other objects such as graphs. An input operation starts from selecting a tool box for the desired input data type and ends with pressing the *Return*

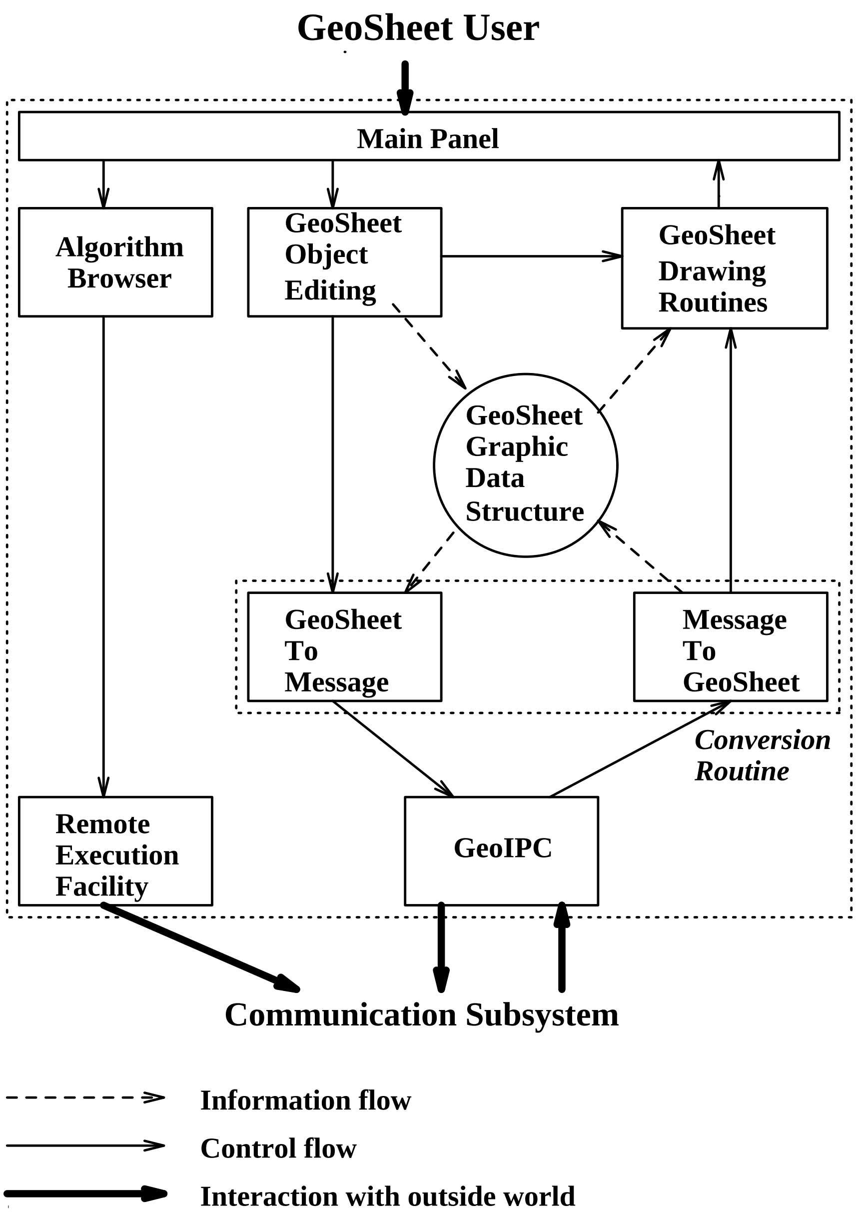

Fig. 3. The *GeoSheet* internal block architecture

button. *GeoSheet* will set the tool box to indicate the current object types. The third is the mechanism for algorithm browsing and execution. This interface is used to browse the available algorithms, select a target algorithm and start the execution. The last kind is the attribute setting boxes to change the current settings of the selected graphic drawing objects. Examples of possible settings include line width, line color, filled pattern, etc.

- *GeoSheet* **Object Editing:** *GeoSheet* object editing routines are invoked by the tool box call back functions in the main panel. These routines support direct manipulation of geometric input/output objects. They include mouse action handling, the intermediate state drawing for interactive input operations, object creation/deletion and other editing related operations.

- *GeoSheet* **Graphics Data Structure:** *GeoSheet* maintains the data structure of all geometric objects in several lists and provides a set of pointers for traversing operations to all components of GeoSheet. Except for the algorithm browser, almost every function in *GeoSheet* relies on this data structure to complete the assigned functions. Since this data structure is heavily dependent on GeoSheet implementation, it is not straightforward to understand this data structure and implement different visualization effects on GeoSheets. One design consideration of *GeoSheet* is to separate the GeoSheet internal structures from programs in order to hide this complexity from algorithm developers.

- **Conversion Routines:** The conversion routines serve as bi-directional bridge between visualization invocation messages and GeoSheet data structure. They consist of message-to-GeoSheet and GeoSheet-to-message conversion operations. The former extracts visualization requests from the messages and creates the display objects for output to GeoSheet Object data structure or invokes graphical input routine for the requested input object. The latter conversion routine is to fill the results obtained from GeoSheet in a message, which gets sent back to the program. In other words, the conversion routines establish the *message protocol*, and perform the transformation between the internal structure of GeoSheet objects and visualization interfaces.

- **Algorithm Browser Graphical User Interface:** Algorithm browser GUI allows the user to overview all available algorithms, and select a particular algorithm for execution. It consists of a graphical user interface connecting to a local or remote file browsing mechanism, a selection mechanism, and interface to remote execution facility. A remote algorithm browser allows browsing of the archives at a remote site and execute any selected algorithm with the remote execution facility.

- **Remote Execution Interface:** The remote execution interface is connected to a mechanism to start a selected algorithm accessible within a computer network. When the user wants to execute a selected algorithm, the algorithm browser will pass the name and the resident site of the algorithm to remote execution interface. Remote execution interface will invoke the remote execution facility in GeoIPC to execute the selected algorithm and pass the required binding connection between *GeoSheet* and the algorithm. Please refer to Section 6.2 for details.

- **Interprocess Communication (IPC) Interface:** The IPC interface uses the process to process communication service supplied by GeoIPC to provide process synchronization and communication for programs and GeoSheet. See Section 6.3.

5.2. On-Line Visualization Invocations

To visualize particular visualization objects, the program needs to invoke visualization interface member functions of the visualization objects, and send GeoSheet messages containing the requested operations, data object type and required arguments. When the messages arrive at GeoSheet, the asynchronous message listening routine in GeoIPC will be activated immediately and pass the incoming messages to the Message-to-GeoSheet conversion routine. The Message-to-GeoSheet conversion routine will first extract the operations and data types from the messages, and perform the appropriate operations as described below.

If the operation is *Graphic_Read*, then the message-to-GeoSheet routine will dispatch the control to the routine capable of performing interactive input operation of the data type in the message. The invoked routine of this particular data type will prompt the user to draw the requested graphic object according to the input sequence of this kind of objects. For example, a polygon object requires the user to input a sequence of points while a point object only needs one click of the mouse. After the user finishes the object drawing and presses *Return* button on GeoSheet main menu, the *Return* callback function will call GeoSheet-to-Message, fill the message buffer with the contents of the just entered input object, and calls GeoIPC to reply to the caller. The graphical input operation is thus finished.

Similarly, the `Graphic_Write` invocation message will reach the Message-to-GeoSheet routine and the object type of the displayed object will be extracted from the message. Message-to-GeoSheet will dispatch the remaining arguments in the incoming message to GeoSheet object creation routine of the output data type. An output object will be created by using the data in the argument list. After the new object is created and added to the GeoObject data structure, a redraw request will be sent to GeoSheet drawing routine and all current GeoSheet objects will be redrawn on the canvas of GeoSheet including the new object. The output operation is then completed.

Other operations are similar and will not be discussed further here.

5.3. Visualization Message Interfaces

In GeoSheet environment, programs and GeoSheets interact through a *visualization message interface*, which defines the representation of the objects passed between the program and GeoSheet. For example, our message interface defines a polygon as a number n of points, followed by n pairs (x, y) representing the x- and y-coordinates of these points, in clockwise or counterclockwise order. This representation only presents the geometric characteristics and is high-level compared to the message format in X11 systems. The message interface also consists of a set of operations to assist programmers to fill and extract the contents from messages. These operations follow the *virtual message buffer* concept in our GeoIPC networking service mechanism, and therefore can manipulate the logically unlimitedly long messages passed by GeoIPC. This abstraction is useful in removing the overhead of flow control processing in case the message length exceeds the limitation of the transporter.

6. Supports for Distributed Visualization

The *GeoSheet* environment supports multiple GeoSheets and multiple algorithms executing simultaneously on distributed homogeneous or heterogeneous machines, while maintaining all features presented in the previous sections. We refer to this type of operations as *distributed execution*.

6.1. Distributed Execution for Program Visualization

Some situations may require users or geometric algorithm developers to execute an algorithm on one site and visualize the computation at the other distributed site. One common case is that the application programs are not available at the local site. If we can execute an algorithm on a remote site directly and visualize the results locally, then we can save the time and storage needed for downloading and installation. Distributed execution is particularly important if the desired program requires special hardware, which may or may not be available at the local site. For example, the program may be implemented on a parallel machine.[1] Algorithm visualization in distributed collaborative environments[2] is yet another application of distributed execution.

To effectively support distributed program visualization, we believe the following facilities are essential.

1. *Remote algorithm browsing.* To execute a program at a remote site, it is desirable to be able to browse all the algorithms on that site. Our scheme is to implement a remote algorithm browser at the site where these algorithms reside. The remote algorithm browser will process requests to list all the files under current or any other directory with authorized access. This kind of operations is provided in the directory service in the ISO application layer, in which similar operations can be found of the `ftp` services. The browser can be implemented as a daemon of the OS at the remote site and respond to any incoming requests from a well known Internet TCP/IP port of that site. To browse the algorithms at a particular site, the GeoSheet will connect to the port reserved exclusively for the algorithm browser daemon, send the requests, such as listing files or changing working directory, and display the returned results.

2. *Remote algorithm execution facility.* The remote execution facility is a mechanism that would enable the user to start a selected algorithm accessible within a computer network. When the user wants to execute a selected algorithm, the algorithm browser will pass the name and the resident site of the algorithm to remote execution facility. Remote execution facility will create a concurrent process to execute the selected algorithm and pass the required binding information to this new process in order to establish the correct connection between GeoSheet and the algorithm. Currently we make use of UNIX *rsh* facility to implement a *delegator* process for handling the above procedures and use a shared memory to pass the binding information. We are designing a remote execution daemon to remove the restriction imposed by *rsh* command, such as requiring an account in order to use *rsh*.

3. *Distributed visualization facility.* Distributed visualization facility governs how the visualization results are to be passed among the distributed programs and active GeoSheets. For instance, after a remote program is initiated, the visual output is required to be sent back to the local GeoSheet for display. Another usage is to distribute the same visualization output to multiple sites as in tele-conference or distributed collaborative environments. (This facility

can be easily supported in our implementation scheme since the communication between the program and GeoSheet is already carried out by using message passing mechanism.)

6.2. Our Distributed GeoSheet Approach

To support distributed execution, we need to tackle an immediate problem, i.e., how a program locates a GeoSheet at the remote site. Two solutions based on TCP/IP protocol are proposed. The first one is whenever a GeoSheet starts, it will announce its IP information such as host name and port number to the public. Whenever a program needs a GeoSheet, the user can input the information manually; therefore the connection can be established. Another scheme is by introducing a centralized control process, *GeoPanel*. Rather than allowing the user to execute GeoSheet or a program manually, this scheme requires all GeoSheets and programs be invoked from *GeoPanel*. Currently our *GeoPanel* is designed with a GUI that allows the user to browse algorithms, select the executable code and start GeoSheet. To the user, *GeoPanel* is a centralized control manager responsible for the management of all available GeoSheets and programs in execution. Any GeoSheet or program started from a *GeoPanel*, it will first be connected to the *GeoPanel*. *GeoPanel* will maintain the information, such as the host machine IP address and the process ID, of the GeoSheet or program. For future communication, the GeoSheets and programs can query *GeoPanel* and get the required information. For example, after a GeoSheet starts, it will report its IP information to *GeoPanel* and *GeoPanel* will maintain this information in a form that can be presented to the user. During program execution, if it requires an additional GeoSheet, it will ask *GeoPanel* for a GeoSheet IP information. *GeoPanel* may prompt the user to select one from the available GeoSheets or create a new GeoSheet directly, and return the IP information of this GeoSheet in order to allow the program to be connected to this GeoSheet. *GeoPanel* also has a name identification mechanism. When a GeoSheet is created with a name string, *GeoPanel* will keep this name string along with IP information. For subsequent GeoSheet allocation operations with a name string, *GeoPanel* will look up the GeoSheet IP information with the supplied name string. By using a name table in *GeoPanel*, all GeoSheets connected to the *GeoPanel* can be located by a name string and the user can ignore the IP information. We refer to the GeoSheets and programs started from the same *GeoPanel* as a *cluster*.

To ensure correct communication of data between the program and GeoSheet, we make every visualization invocation a remote procedure call (RPC) to transfer the control and pass the data from the program to GeoSheet. Our RPC differs from a conventional approach in that the data represented in our RPC are geometric objects. That is, after the control is passed to the called routine, RPC will pass geometric objects directly to the callee. For example, a polygon visualization invocation will invoke the polygon drawing routine in GeoSheet. However, after the GeoSheet polygon drawing routine is invoked, our RPC will extract the polygon as input parameter to this routine. Therefore both the caller and callee can be thought of as if they would operate on an abstract geometric object representation layer. This layer will make the visualization task more clear and independent from details of the communication transporter.

Besides the connection between the program and GeoSheet, the management of the distributed programs and GeoSheets is also important. Since there may be multiple GeoSheets and several remote programs active among the distributed sites, it is required to provide utilities to assist the user in monitoring the current system status and perform remote execution effectively. *GeoPanel* is also designed to take care of this management as well. The user may list all the active GeoSheets and connect remote site algorithm browsers through *GeoPanel* GUI.

Compared to other distributed visualization scheme, the most significant advantage in our approach is that the invocation messages can be directly generated from the program. This provides a much more flexible tool compared to those off-line data visualization schemes such as Ref. [21]. Besides, our geometric representation message layer can effectively reduce the number of messages. The X11 client/server messages are of finer grain size and will produce a larger number of messages for remote execution compared to our scheme. In contrast, Mosaic messages deal with large grain computation and are usually too big to be used for interactive programming aid. With the aid of object-oriented methodology, the grain size of invocation message can be made at the statement level, or any C++ object level which means a much more flexible structure than the above schemes. This scheme can reflect the execution and the control flow as closely as algorithm designers wish.

6.3. GeoIPC

We have designed a *GeoIPC* runtime library to realize our distributed GeoSheet scheme. All components in GeoSheet are required to integrate GeoIPC into their functions.

GeoIPC supports the connection establishment, communication and synchronization for *GeoSheet*. Its functions include connecting distributed GeoSheets and algorithm executions; locating the GeoSheet specified in the `QuerySheet` operation; and the message communication utility supporting virtual buffer of logically unlimited length.

The procedures to establish connections between distributed algorithm executions and GeoSheets are complex because the interactions of the multiple active GeoSheets and algorithm executions may be complicated. For example, a remote algorithm may be started by either *rsh* or remote execution daemon when responding to the requests from GeoSheet or *GeoPanel*. The GeoSheet to which this algorithm is to be connected will be different. If an algorithm is started by a GeoSheet then it is required to be connected to its starting GeoSheet. If it is from a *GeoPanel*, then it can only be connected to a GeoSheet instantiated by this *GeoPanel*. Otherwise if it is started by *rsh* or from the command line, then it will prompt the user to type the GeoSheet binding information at the text command line. Therefore one of the most important functions of GeoIPC linked to an algorithm is to distinguish the starting scheme and establish the correct connections accordingly.

We currently use a *delegator* function to establish the initial connection. Whenever an algorithm execution is started, either locally or remotely, the execution facility will invoke the *delegator* to put the binding information of the *GeoPanel* or GeoSheet which starts the algorithm to a shared storage, then start the algorithm. When an algorithm is started, its GeoIPC initialization routines will first check whether such shared storage is available. If so, the algorithm is started by GeoSheet or *GeoPanel*; otherwise it is started from the command line. Following this kind of protocol, GeoIPC will communicate with the starting GeoSheet or *GeoPanel*, and establish the connection.

Locating GeoSheets specified by `QuerySheet` is another important function of GeoIPC. It is important to have GeoSheets shared among algorithms in distributed environments. We rely on *GeoPanel* to resolve the name identification problem. When an algorithm queries a logical name of a GeoSheet, its GeoIPC will send the name string to its connecting *GeoPanel* where all the information of the GeoSheets belonging to the same cluster is available. *GeoPanel* will look up the name and return the binding information to the requesting algorithm. Our current scheme, however, only allows an algorithm to query GeoSheets in the same *GeoPanel* cluster.

GeoIPC also provides process synchronization and communication for programs and GeoSheet. Central to GeoIPC is the *virtual buffer* concept which is based on the TCP/IP protocol and supports communication/synchronization functions including listening to asynchronous incoming messages and sending messages between different machine sites. We implement GeoIPC on top of TCP/UDP, and thus it can be across the Internet and is virtually applicable to all computer networks.

7. Current Implementation Status

Currently our visualization tool, *GeoSheet*, is built on top of Xfig. Some modifications to the source code of Xfig were necessary to meet our needs. We add a TCP/IP message interface and geometric message protocol conversion modules in order to support message driven input/output style. The operations are mainly realized by a set of messages to and from Xfig object conversion routines, which after extracting necessary information for output pass the contents of selected Xfig objects to Xfig or in the case of input, convert the input Xfig objects into a return message. Several attribute setting and user query supporting functions that can be triggered from messages are also available. At the moment, only two-dimensional display is supported, but it is designed so that the output can be easily directed to different display devices, including 3-dimensional displays. We are now designing and implementing a new 3-D visualization subsystem, *Geo3DSheet*, based on this message protocol. This tool can be used as a substitute and be used to demonstrate the transparency of converting visualized geometric algorithms from 2-D to 3-D.

The main purpose of the reuse of the Xfig code is that it not only saves our prototyping time but also provides a set of reliable 2D drawing tools that meet our needs. This release has been heavily used in testing our ideas of visualization, direct manipulation, and algorithm invocation, and serves very well as a major platform in distributed environments. Our results indicate the message interface used in invoking *GeoSheet* visualization operations is not related to either Xfig internal structures or X11 display technical metrics.

Visualization classes have been implemented on three existing geometric classes. Among them is *GeoLEDA* which is derived from LEDA classes. To demonstrate the easiness of transforming non-visualization oriented LEDA program, we remove the window display part of several LEDA examples and port them to *GeoSheet*. Our experience indicates such transformation can be done within a few days. The results produced so far are quite satisfactory, and we hope that the computational geometry research community and others will find this tool useful when they develop their algorithms.

8. Concluding Remarks

GeoSheet is an interactive visualization tool for visualizing geometric algorithms in distributed environments. It provides features such as interactive visualization of program states for debugging, high-level graphical input/output manipulation facilities for geometric objects, reuse of existing data structures and algorithms implementation, and more importantly distributed executions for heterogeneous machines at different sites.

Several advantages of using this tools are as follows. First, the almost identical Xfig graphical interface supports a widely used graphical objects manipulation mechanism. Both the input and output data can be visualized, printed or exported for other purposes. Since the file can be saved and reloaded, it can reduce time for testing and verification of the correctness of the implementation of an algorithm. Second, the introduction of message driven interface allows separation of the execution of an algorithm from the visualization subsystem. With such separation, the algorithm implementor can concentrate on algorithmic operations and neglect the complex portion of visualization procedures. Furthermore visualization can be done at a remote site and possibly on machines different from the one that executes the algorithm. In other words, the tool is distributed, and device independent. By controlling display messages, one can interactively execute an algorithm and have its intermediate output displayed with *GeoSheet*. This feature is useful in interactive debugging.

We use object-oriented programming methodology to construct the visualization interface. By deriving from traditional data type and algorithm libraries, our abstract GeoObject super-classes have been proven to be easy to use, easy to construct, and highly portable.

Acknowledegement

This work was supported by the Office of Naval Research under the Grant No. N00014-93-1-0272. The authors are grateful to Dave Fisher for his initial work on the conversion routines of *GeoSheet*, Laurent H. Lin for his work on *GeoPanel*, Kiyoko Aoki for her implementation and documentation of polygon visibility algorithm, and Christopher Kush for testing and other numerous tasks involved in the overall project.

References

1. G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin, Redwood, CA, 1989.
2. V. Anupam, C. Bajaj, D. Schikore and M. Schikore, "Distributed and Collaborative Visualization", *Computer*, July 1994, pp. 37-43.
3. M. Accetta, et al., "SHASTRA: A Distributed and Collaborative Design Environment", *Animation of Geometric Algorithms: A Video Review*, June 1992, pp. 12-14.
4. J. E. Baker, I. F. Cruz, G. Liotta and R. Tamassia, "A New Model for Algorithm Animation Over the WWW," *ACM Comput. Surveys*, (27,4) 1995, pp. 568-572.
5. J. E. Baker, I. F. Cruz, G. Liotta and R. Tamassia, "Algorithm Animation Over the World Wide Web," *Proc. Int. Workshop on Advanced Visual Interfaces,* 1996, pp. 203-212.

6. J. E. Baker, I. F. Cruz, G. Liotta and R. Tamassia, "Animating Geometric Algorithms over the Web," *Proc. 12th Annu. ACM Sympos. Comput. Geom.* June 1996, pp. C3-C4.
7. G. Barequet, S. S. Bridgeman, C. A. Duncan, M. T. Goodrich and R. Tamassia, "Classic Computational Geometry with GeomNet," *Proc. Annu. ACM Sympos. Comput. Geom.* June 1997, pp. 412-414.
8. M. H. Brown, *Algorithm Animation*, MIT Press, Cambridge, MA. 1988.
9. M. H. Brown, "Zeus: A System for Algorithm Animation and Multi-view Editing," SRC Tech. Report, Digital Equipment Corporation, Feb. 1992.
10. M. H. Brown and J. Hershberger, "Color and Sound in Algorithm Animation," SRC Tech. Report, Digital Equipment Corporation, Aug. 1991
11. M. H. Brown and J. Hershberger, "Animation of Geometric Algorithms: A Video Review," Editors, 87a-b, Digital Equipment Corporation, June 1992.
12. M. H. Brown and J. Hershberger, "Video Review," *Proc. 9th Symp. on Computational Geometry,*, June 1993, pp. 391.
13. M. H. Brown and R. Sedgewick, "A System for Algorithm Animation," *Computer Graphics,* 18,3, July 1984. pp. 177-186.
14. M. H. Brown and R. Sedgewick, "Techniques for Algorithm Animation," *IEEE Software,* (2,1) Jan. 1985, pp. 28-39.
15. P. J. de Rezende and W. R. Jacometti, "GeoLab: An Environment for Development of Algorithms in Computational Geometry". *Proc. 1st Int'l Computational Geometry Software Workshop*, Geometry Center, MN, Jan. 18-20, 1995; see also *Proc. 9th Symp. on Computational Geometry*, May 1993, pp. 401-402.
16. H. Edelsbrunner and E. P. Mücke, "Three-Dimensional Alpha Shapes," *ACM Trans. on Graphics,* (13,1) Jan. 1994, pp. 43-72.
17. M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.
18. P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack, "A Workbench for Computational Geometry," *Algorithmica,* April 1994, pp. 404-428.
19. S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmic*, 2, 1987, pp. 153-174.
20. R. Gruia-Catalin and K. C. Cox, " A Taxonomy of Program Visualization Systems," *Computer*, Dec. 1993, pp. 11-24.
21. A. J. Hanson, T. M. Munzner, and G. Francis, "Interactive Methods for Visualizable Geometry," *Computer*, (27,7) July 1994, pp. 73-83.
22. D. T. Lee and A. K. Lin, "Computational Complexity of Art Gallery Problems," *IEEE Trans. Infor. Theory,* (IT-32,2) March 1986, pp. 276-282.
23. K. Mehlhorn and S. Näher, "LEDA — A Library of Efficient Data Types and Algorithms," Lecture Notes in Computer Science, Springer-Verlag, Vol 379, 88-106, 1989.
24. T. Munzner, L. Stuart, and M. Phillips, "GeomView User Manual", *Geometry Center software*, Nov. 1993.
25. S. Näher, "LEDA – A Library of Efficient Data Types and Algorithms," Max-Planck-institut für informatik, Saarbrücken, 1992.
26. J. O'Rourke, *Art Gallery Theorems and Algorithms*, New York, Oxford, Oxford University Press, 1987.

27. P. Schorn, "An Object Oriented Workbench for Experimental Geometric Computation," *Proc. 2nd Canadian Conference in Computational Geometry*, Ottawa, August 6-10, 1990, pp. 172-175.

28. C. F. Shen, "The λ-geometry Steiner Minimal Tree Problem and Visualization," Ph.D. Dissertation, Dept. of Electrical and Comput. Engineering, Northwestern Univ., Feb. 1997.

29. B. V. Smith, *The Xfig User Manual*, 1993.

30. J. T. Stasko, "Animating Algorithms with XTANGO," SIGACT News, (23,2) Spring 92, pp. 67-72.

31. A. Tal and D. Dobkin, "Visualization of Geometric Algorithms," *IEEE Transactions on Visualization and Computer Graphics,* (1,2) June 1995, pp. 194-204.