Project Report : Extensions to GeomLib

Baolin Yang
Department of Computer Science
Brown University

May 14, 1998

Part I : A Mobile Agent Model for Remote Geometric Computing.

Part II : Testing, Revising and Using the TripartiteEmbeddedPlanarGraph Java Class.

Extensions to GeomLib

Algorithm Engineering for a Geometric Computing Library

A Mobile Agent Model for Remote Geometric Computing; Testing, Revising and Using the TripartiteEmbeddedPlanarGraph Java Class

by Baolin Yang
Deparment of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University.

[illegible] 5/18/98

Signature(Roberto Tamassia/Franco P. Preparata, Advisor) Date

A Mobile Agent Model for Remote Geometric Computing

Baolin Yang

May 14, 1998

1 Introduction

One aspect which is of increasing importance to make geometric computing libraries relevant to practical applications is the distribution of the components of an application system. The simplest case is the client-server paradigm. A geometric server could support certain services made available to client applications. In this report, we shall discuss a new communication paradigm, the mobile agent, for remote geometric computing.

The central organizing principle of today's computer communication networks is remote procedure calling (RPC). Conceived in the 1970s, the RPC paradigm views computer-to-computer communication as enabling one computer to call procedures in another. The salient characteristic of remote procedure calling is that each interaction between the user computer and the server entails two acts of communication, one to ask the server to perform a procedure, and another to acknowledge that the server did so. Thus ongoing interaction requires ongoing communication.

For remote geometric computing using libraries, however, an alternative an better approach appears to be remote programming (RP). The salient characteristic of remote programming is that a user computer and a server can interact without using the network once the network has transported an agent, an independent software program which runs on behalf of a network user, between them. Thus ongoing interaction does not require ongoing communication.

In this report, we first propose an easy-to-implement communication paradigm using Remote Method Invocation (RMI) method. In this model, a client can obtain a geometric computing object from geometric libraries. This approach is compared with a traditional Client-Server paradigm that geometric libraries can be accessed and provide service remotely through library interfaces. Through the analysis of the above two communication paradigms, our preference is given to the first approach due to the fact that it is easy to implement and it does not require communication once a geometric computing object has been transported to user.

Then we will introduce a mobile agent communication paradigm for remote geometric computing. A mobile agent can do work on behalf of a network user on server machines while in the easy-to-implement paradigm the geometric computing object needs to be executed on a client machine. This feature is desirable in many applications. The mobile agent paradigm has many advantages over other paradigms. We shall discuss these in detail in this report.

In our discussion, we assume that interfaces are separated from their implementations in the geometric computing library we are considering, and they are available on the client side as well as the server side. Note that clients need to have these interface definitions anyway in order to know what he can do.

The remaining part of the paper is organized as follows. In section 2 we propose the easy-to-implement communication paradigm and discuss its properties. In section 3 we introduce the concept of mobile agent. In section 4 we give the design of our mobile agent communication paradigm for geometric computing using libraries. We explain the implementation details in section 5. In section 6 we give some concluding remarks.

2 A Easy-to-Implement Communication Paradigm for Remote Geometric Computing

We first propose a easy-to-implement communication paradigm for remote geometric computing using libraries. The key part of this paradigm is to implement an object server on the server side. The functionality of the object server is instantiating geometric computing objects with parameters specified by callers. The interface for the object server is also designed to be the Remote Method Invocation interface. On the client side, one can contact the object server and obtain the requested geometric computing objects. Then the client can perform his task locally using the geometric computing objects. No further communication is needed.

In the traditional Client-Server approach, one usually extends the interfaces of geometric computing libraries such that they support remote procedure calls or remote method invocation. This design means that we need to build the Client-Server model inside the geometric computing libraries by extending all interfaces of the libraries to provide remote services.

Here we give a comparison of the above two approaches in the following.

The first model is easy to implement. It also has the advantage that the implementation of the Client-Sever model is essentially separated from the libraries themselves. In a Java implementation of this model, the only change we need to

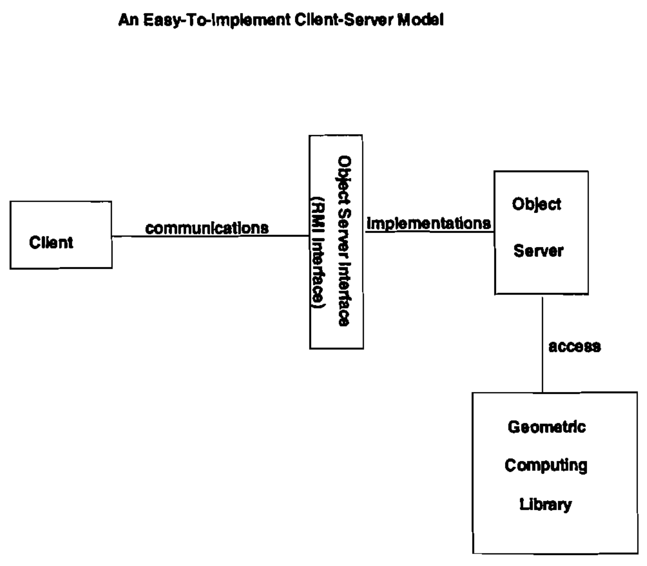
An Easy-To-Implement Client-Server Model
Client
communications
Object Server Interface (RMI Interface)
implementations
Object Server
access
Geometric Computing Library

Figure 1: An easy-to-implement Client-Server model for geometric computing with libraries (GeomLib).

make to the libraries is to let the interfaces implement Java.io.Serializable which is an empty interface.

The first model is also efficient. After the client has obtained a geometric computing object from the server, the client can run the object locally and no further communication is needed.

The first model has the restriction that the application needs to be ran on the client side. This restriction is not desirable in many applications in network computing.

In the second model, clients can run their applications on the server side using the geometric computing library. However, this model is inherently coupled with the library, which is not a desirable feature in implementation and extension. More importantly, this model is also not efficient because each method call to the library means a communication between the client and the server. Note that in the first model, the number of communications is approximately equal to the number of objects one needs to use.

The above two Client-Server models both have their drawbacks and are not satisfying. Nevertheless, the first model, which is very easy to implement, is very favorable in applications where client machines are powerful enough to run geometric computing algorithms.

3 The Concept of A Mobile Agent

While it means a variety of things to a variety of people, an agent in this report is defined as an independent software program which runs on behalf of a network user. The agent can be dispatched from the client side and bring client's computing tasks to the servers. On the server side, the agent performs the tasks utilizing resource of the server. In our case, the resource on the server side is the geometric computing libraries. An agent may work even after the user is disconnected from the network. Many examples of agent systems exist, and they are receiving much attention on the World Wide Web.

A mobile agent can travel to multiple locations on the network and performs the corresponding tasks at each location. This mobility greatly enhances the productivity of each computing element in the network. This uniquely powerful computing environment is well suited for open geometric computing library applications.

Mobile agent programming has the advantages of being efficient and secure. It is efficient because an agent can make best use of the resource at one site and perform relevant tasks at the site. This reduces the number of communications over the network. It is secure because agents can carry credentials with them as they travel and these credentials can be authenticated by the agent servers at points in the network. One has the flexibility in implementing security measures in the authentication process.

Mobile agent programming also has the advantage of allowing generic programming in server implementations. An agent server only needs to provide a platform for agents to execute and provide the connection between agents and local resources. Hence agent server implementation is isolated from specific client applications and the agent server can be the same for different client applications.

In the following we want to present a mobile agent specializing in remote geometric computing. A typical scenario is that a client wants to perform some geometric computing tasks that need to utilize remote geometric computing libraries. The client wants to send an agent to the servers of the libraries to perform the tasks and report the results to him. The mobile agent model may also be used to dynamically partition code between clients and servers.

4 A Mobile Agent Paradigm for Geometric Computing with Open Libraries

In our geometric computing mobile agent paradigm, we have an agency and a server (or servers). An agency is responsible for dispatching agents and receiving reports from agents. A server provides a place as a platform for agents to perform its

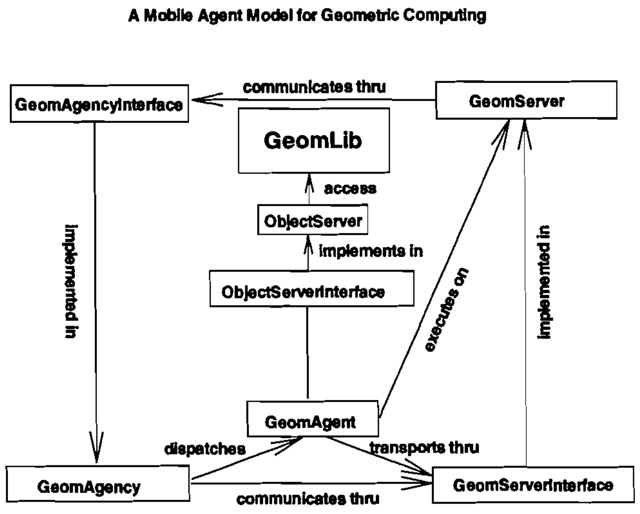

Figure 2: A mobile agent model for geometric computing with libraries (GeomLib).

tasks. By implementing callbacks using Java's RMI, we set peer-to-peer relationships between interacting client and server objects, since simply having remote objects and being able to transport remote objects is not sufficient for complex interactions. We also incorporate an object server into this paradigm. The mobile agent accesses the libraries through the interface of the object server. The functionality of the object server is instantiating geometric computing objects with parameters specified by callers. See Fig. 2 for an illustrative diagram of the model.

An important functionality of a server is managing the resources in geometric computing libraries using the object server. And the interface to the server consists of generic "newObject" methods, which is implemented in the object server to instantiate various geometric objects. In this paradigm, we isolate the implementation of geometric computing libraries from agents, which is a desirable feature since modifications of the implementation of geometric computing libraries will require no change whatsoever on the mobile agent implementations.

As in other agent systems, a server needs to provide a platform for agents to run their jobs. The platform in our design is called Place, borrowed from the concept in Odyssey developed by General Magic. The functionalities of a Place will be explained in detail later.

5 Implementation Details of Geometric Computing Agent

We briefly explain the implementation details of our geometric computing agent by describing the following classes in our implementation:

Engine : an execution engine that creates places, where agents can run their tasks and be transported. It also establishes a GeomAgency that is responsible for dispatching agents and receiving reports from agents.

Process : an abstract class that gives the basic functionalities of two main running threads : the agent process and the place process.

Place : a process that can accept agents and sponsor threads to run the agents. It is implemented as a thread running on the server side, waiting for agents to come.

Agent : a process that can run in places and travel from places to places. Only the basic functionalities are implemented in this class, which serves as the super class for Worker and GeomAgent, where more functionalities of a geometric computing agent are implemented.

Worker : an agent that can perform tasks on the server side. To be more specific, it finds the methods in a GeomAgent corresponding to different tasks, that a client wants to perform, and execute the methods in places. It also provides the reporting mechanism for an agent to send reports to the GeomAgency.

GeomAgent : a geometric computing agent that brings the geometric computing tasks with it, performs the tasks on the sites that have relevant resources. In application, a client can write their geometric computing program in this class, as tasks to be performed.

GeomAgency : an agency that is responsible for dispatching agents, receiving and processing reports from agents, and controlling further actions of agents.

GeomServer : a server that provides the communication mechanism including the transport of agents and the set-up of call-back interface.

ObjectServer : a server that instantiates geometric computing objects to be used by agents. It instantiates the objects with the parameters specified by the caller.

An Agent is transported into a server by calling GeomServer's "transferIn" method. The server Engine will then inform the Place at the site to accept the Agent by calling its "entry" method. The thread of the Place, which is waiting to accept Agents, will resume its execution and create a thread and a threadgroup to sponsor the Agent. Thus the Agent is activated in the Place and its tasks are performed in the Place. After the agent performs all its tasks, it will move on to the next location by calling the "transferOut" method in Engine. The server Engine informs the Place to let the Agent exit and stops the thread execution of the exiting Agent thread.

The control of the path of an Agent is specified in its task list, which includes the location of the sites the Agent wants to visit and the tasks it wants to perform at the sites. An Agent can modify its task list on the path, and one can implement an intelligent Agent that knows what to do and where to go on its own.

6 Concluding Remarks

In the report, we discuss different client-server models for remote and network geometric computing. In particular, we propose an easy-to-implement communication paradigm, that can be implemented using Java, and a mobile agent communication paradigm that is very suited for remote geometric computing using open libraries. The model we propose allows for efficient geometric computing applications on the network.

This paradigm can be applied to partition code dynamically between clients and servers. It can also be easily extended to allow for distributing the implementations of libraries at different locations, which is a desirable feature for collaboration purposes. It also has the advantage of being able to implement desired security measures in accepting Agents to Places.

References

[1] D. M. Chess, C. G. Harrison, and A. Kerschenbaum, *Mobile Agents: Are they a good idea?*, IBM Research Report, RC 19887, 1994.

[2] D. T. Chang and D. B. Lange, *Mobile Agents: A New Paradigm for Distributed Object Computing on the WWW*, in Proceedings of the OOPSLA96 Workshop: Toward the Integration of WWW and Distributed Object Technology, October 1996.

[3] J. White, *Mobile Agents White Paper*, General Magic, URL= http://www.genmagic.com/agents/Whitepaper/whitepaper.html

[4] *Mobile Agent Computing - A White Paper*, URL= http://www.meitca.com/HSL/Projects/Concordia/MobileAgentsWhitePaper.html

[5] *Aglets: Mobile Java Agents, IBM Tokyo Research Lab*, URL= http://www.ibm.co.jp/trl/projects/aglets

[6] D. T. Chang and D. B. Lange, *Programming Mobile Agents in Java*, URL= http://www.trl.ibm.co.jp/aglets/

[7] M. Strasser, J. Baumann, and F. Hohl, *MOLE: A Java Based Mobile Agent System*, in Proceedings of the European Conference on Object Oriented Programming, 1996.

[8] *Object Serialization for Java*, Javasoft Corporation, URL= http://chatsubo.javasoft.com/current/serial/index.html

[9] *Remote Method Invocation for Java*, Javasoft Corporation, URL= http://chatsubo.javasoft.com/current/rmi/index.html

[10] J. Gosling, F. Yellin, and The Java Team, *Java API Documentation Version 1.0.2 - Class ClassLoader*, URL=http://java.sun.com/products/JDK/1.0.2/api/

[11] D.S. Milojicic, M. Condict, F. Reynolds, D. Bolinger, and P. Date, *Mobile Objects and Agents*, in Proceedings of the Second USENIX Conference on Object Oriented Technologies and Systems (COOTS), Toronto, Canada, June 1996.

Appendix : Java RMI Implementation Notes

A. Bootstrapping the Client

For the RMI runtime to be able to download all the classes and interfaces needed by a client application, a bootstrapping client program is required which forces the use of a class loader (such as RMI's class loader) instead of the default class loader. The bootstrapping program needs to:

Create an instance of the RMISecurityManager or user-defined security manager. Use the method RMIClassLoader.loadClass to load the class file for the client. The class name cannot be mentioned explicitly in the code, but must instead be a string or a command line argument. Otherwise, the default class loader will try to load the client class file from the local CLASSPATH.

Use the newInstance method to create an instance of the client and cast it to Runnable. Thus, the client must implement the java.lang.Runnable interface. The Runnable interface provides a well-defined interface for starting a thread of execution. Start the client by calling the run method (of the Runnable interface).

For example:

```
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;

public class LoadClient
```

```
{
        public static void main()
        {
                System.setSecurityManager(new RMISecurityManager());

                try {
                        Class cl = RMIClassLoader.loadClass("myclient");
                        Runnable client = (Runnable)cl.newInstance();
                        client.run();
                } catch (Exception e) {
                        System.out.println("Exception: " + e.getMessage());
                        e.printStackTrace();
                }
        }
}
```

In order for this code to work, you need to specify the java.rmi.server.codebase property when you run the bootstrapping program so that the loadClass method will use this URL to load the class. For example:

```
java -Djava.rmi.server.codebase=http://host/rmiclasses/ LoadClient
```

Instead of relying on the property, you can supply your own URL:

```
Class cl = RMIClassLoader.loadClass(url, "myclient");
```

Once the client is started and has control, all classes needed by the client will be loaded from the specified URL. This bootstrapping technique is exactly the same technique Java uses to force the AppletClassLoader to download the same classes used in an applet.

B. RMI Through Firewalls Via Proxies

The RMI transport layer normally attempts to open direct sockets to hosts on the Internet. Many intranets, however, have firewalls which do not allow this. The default RMI transport, therefore, provides two alternate HTTP-based mechanisms which enable a client behind a firewall to invoke a method on a remote object which resides outside the firewall.

To get outside a firewall, the transport layer embeds an RMI call within the firewall-trusted HTTP protocol. The RMI call data is sent outside as the body of an HTTP POST request, and the return information is sent back in the body of the HTTP response. The transport layer will formulate the POST request in one of two ways:

1. If the firewall proxy will forward an HTTP request directed to an arbitrary port on the host machine, then it is forwarded directly to the port on which the RMI server is listening. The default RMI transport layer on the target machine is listening with a server socket that is capable of understanding and decoding RMI calls inside POST requests.

2. If the firewall proxy will only forward HTTP requests directed to certain well-known HTTP ports, then the call will be forwarded to the HTTP server listening on port 80 of the host machine, and a CGI script will be executed to forward the call to the target RMI server port on the same machine.

In configuring the server, the host name should not be specified as the host's IP address, because some firewall proxies will not forward to such a host name.

1. In order for a client outside the server host's domain to be able to invoke methods on a server's remote objects, the client must be able to find the server. To do this, the remote references that the server exports must contain the fully-qualified name of the server host. Depending on the server's platform and network environment, this information may or may not be available to the Java virtual machine on which the server is running. If it is not available, the host's fully qualified name must be specified with the property java.rmi.server.hostname when starting the server.

For example, use this command to start the RMI server class ServerImpl on the machine chatsubo.javasoft.com:

```
java -Djava.rmi.server.hostname=chatsubo.javasoft.com ServerImpl
```

2. If the server will not support RMI clients behind firewalls that can forward to arbitrary ports, use this configuration: 1. An HTTP server is listening on port 80. 2. A CGI script is located at the aliased URL path /cgi-bin/java-rmi. This script: Invokes the local Java interpreter to execute a class internal to the transport layer which forwards the request to the appropriate RMI server port. Defines properties in the Java virtual machine with the same names and values as the CGI 1.0 defined environment variables. An example script is supplied in the RMI distribution for the Solaris and Windows 32 operating systems. Note that the script must specify the complete path to the java interpreter on the server machine.

Testing, Revising and Using the TripartiteEmbeddedPlannarGraph Java Class

Baolin Yang

May 14, 1998

1 Introduction

For correctness checking, GeomLib geometric computing library incorporates hierarchical checkers in order to enhance reliability. Program checkers are integrated within the library to boost confidence in the algorithms. Checkers may also provide valuable insight into the corresponding algorithms.

The objective of this project is to implement a tester for embedded planar graph implementations for GeomLib and to use the tester to fully test the TripartiteEmbeddedPlannarGraph implementation, where a tripartite graph is used for a graph's internal representation.

The tester consists of following parts:

1. the initial test,
2. the Main test,
3. the connectivity test,
4. the planarity test
4. the two-dimensional grid test,
5. the triangulation of two chains test.

In the testing of the TripartiteEmbeddedPlannarGraph implementation, some problems are found. Many of them are related with a subtle implementation mistake, i.e. the ordering of edges connecting a face vertex and incident edge vertices is wrong in some cases. Efforts have been taken to revise the implementation and fix all the problems. The revised version of the implementation can pass the tester and has no known bug.

I also implemented a biconnectivity augmentation algorithm using the TripartiteEmbeddedPlannarGraph as its underlying data structure. Besides being another test for the TripartiteEmbeddedPlannarGraph implementation, the algorithm is useful in application.

2 The Tester

In the following, I explain the components of the tester in detail.

In the initial test, we check if the initial graph has one and only one vertex, and if the number of faces and edges is correct.

In the Main test, we test all the methods in the implementation. We first try to use all the constructing methods to draw a complicated embedded planar graph. Then we try to access the vertices, edges, and faces. Finally we try to undo the graph using the undo methods.

In the connectivity test, we test if the implementation can determine correctly whether a deletion leaves the graph disconnected.

In the planarity test, we test if the implementation can determine correctly whether an insertion violates planarity.

In the two-dimensional grid test, we draw a two-dimensional rectangular grid using the implementation. We can control the size of the grid by setting parameters. We can use this method to generate a graph of an arbitrary size.

In the triangulation of two chains test, we triangulate the region between two chains. The length of the chain is also controlled by a parameter, and we can use this method to generate a graph of an arbitrary size.

The non-trivial bug we find in the testing of the tripartite implementation occurs when one attaches many vertices to one vertex. In this case, the ordering of edges incident on the face is wrongly represented in the tripartite graph. Weird errors will occur if we proceed with this wrong ordering.

3 Revision of the TripartiteEmbeddedPlannarGraph Implementation

We indeed find a lot of points that need to be fixed in the tripartite implementation of EPG. The revision together with the original implementation will be listed in the following. We also briefly explain the reasons for the changes. Some of them are subtle.

`In Method "_duplicateFaces(Vertex a)":`

Line

```
"while(i != startIndex && avec.size() > 1) {"
```

—replaced—

```
"while(i != startIndex) {"
```

Reason for Change: Problems occur when one wants to delete a vertex that has only one incident edge. In this case, one can always delete the vertex, and it is not necessary to execute the while loop. In the original implementation, it still executes the while loop, resulting in an error.

In Method "_insertEdgeHelper(Vertex u, Edge eu, Order orderu, Vertex v, Edge ev, Order orderv, Object info)":

Lines

```
"puEdge = _tripartite.nextIncidentEdge(pu, uPrevEdge);
 nuEdge = _tripartite.prevIncidentEdge(nu, uNextEdge);
 pvEdge = _tripartite.nextIncidentEdge(pv, vPrevEdge);
 nvEdge = _tripartite.prevIncidentEdge(nv, vNextEdge);"
```

—replaced—

```
"puEdge = _tripartite.prevIncidentEdge(pu, uPrevEdge);
 nuEdge = _tripartite.nextIncidentEdge(nu, uNextEdge);
 pvEdge = _tripartite.prevIncidentEdge(pv, vPrevEdge);
 nvEdge = _tripartite.nextIncidentEdge(nv, vNextEdge);"
```

Reason for Change: For the consistency of inserting edges, we need to make these modifications. Otherwise, the order of edges incident on a face can become messed up.

Lines

```
"Edge bottom = _tripartite.insertEdge(vvertu, uPrevEdge, Order.AFTER,
newEvert, etob, Order.BEFORE, Container.NULL);
  ...
 Edge top = _tripartite.insertEdge(vvertv, vPrevEdge, Order.AFTER,
newEvert, atoe, Order.BEFORE, Container.NULL);"
```

—replaced—

```
"Edge bottom = _tripartite.insertEdge(vvertu, uPrevEdge, Order.AFTER,
newEvert, etob, Order.AFTER, Container.NULL);
  ...
 Edge top = _tripartite.insertEdge(vvertv, vPrevEdge, Order.AFTER,
newEvert, atoe, Order.AFTER, Container.NULL);"
```

Reason for Change: For the consistency of inserting edges, we need to make these modifications. Otherwise, the order of edges incident on a face can become messed up.

`In Method "removeVertex(Vertex v)":`

I consider the case when vertex v only has one incident edge. This case will cause problem for the original implementation.

`In Method "splitEdge(Edge e, Object info)":`

We need to consider whether the edge to be split is directed or not. If it is, we need to keep the direction in the split edges. I add a condition

```
"if (_tripartite.isDirected(ea))"
```

in Lines

```
 "Edge newedge1=null, newedge2=null;
  if (_tripartite.isDirected(ea)) {
    if (evert == _tripartite.destination(ea)) {
      newedge1 = _tripartite.insertDirectedEdge(fa, ea, Order.AFTER,
evert, ea, Order.AFTER, Container.NULL);
      newedge2 = _tripartite.insertDirectedEdge(evert, eb, Order.BEFORE,
fb, eb, Order.BEFORE, Container.NULL);
    } else {
      newedge1 = _tripartite.insertDirectedEdge(evert, ea, Order.AFTER,
fa, ea, Order.AFTER, Container.NULL);
      newedge2 = _tripartite.insertDirectedEdge(fb, eb, Order.BEFORE,
evert, eb, Order.BEFORE, Container.NULL);
    }
  } else {
     newedge1 = _tripartite.insertEdge(evert, ea, Order.AFTER,
fa, ea, Order.AFTER, Container.NULL);

     newedge2 = _tripartite.insertEdge(evert, eb, Order.BEFORE,
fb, eb, Order.BEFORE, Container.NULL);
  }"
```

—replaced—

```
"Edge newedge1 = _tripartite.insertEdge(evert, ea, Order.BEFORE,
fa, ea, Order.AFTER, Container.NULL);
  Edge newedge2 = _tripartite.insertEdge(evert, eb, Order.AFTER,
fb, eb, Order.BEFORE, Container.NULL);"
```

Line

```
"Edge newEdge = _tripartite.splitVertex(evert, newedge1, newedge2,
Container.NULL);"
```

—replaced—

```
"Edge newEdge = _tripartite.splitVertex(evert, newedge2, newedge1,
Container.NULL);"
```

We add Lines

```
"Edge temp=null;
  if (_tripartite.prevIncidentEdge(ends[1], ea) != newEdge)
    temp = _tripartite.prevIncidentEdge(ends[1], ea);
  else
    temp = _tripartite.nextIncidentEdge(ends[1], ea);

  try {
    if (_tripartite.destination(temp) == ends[1]) {
      _tripartite.setDirectionFrom(newEdge, ends[1]);
    } else {
      _tripartite.setDirectionTo(newEdge, ends[1]);
    }
  } catch (InvalidEdgeException xcp) {
    // Should not be here.
  }"
```

Reason for Change: The original implementation did not consider the direction of the edge to be split and the directions of the split edges. Also, since the structure of the internal TripartiteEPG is different in the modified version, necessary changes are also made here.

```
In Method "attachVertex(Vertex v, Edge e, Order order,
Object vertexInfo, Object edgeInfo)":
```

Line

```
"if (order == Order.AFTER)"
```

—replaced—

```
"if (order == Order.BEFORE)"
```

Line

```
"else if (order == Order.BEFORE)"
```

—replaced—

```
"else if (order == Order.AFTER)"
```

Line

```
"Edge top = _tripartite.attachVertex(newEvert, bottom, order, Container.NULL,
Container.NULL);"
```

—replaced—

```
"Edge top = _tripartite.attachVertex(newEvert, bottom, order, Container.NULL,
Container.NULL);"
```

I add a condition

```
"if (order == Order.BEFORE)"
```

in Lines

```
"if (order == Order.BEFORE) {
    Edge prevFaceEdge = _tripartite.insertEdge
(newEvert, bottom, Order.AFTER, fvert, faceEdge, Order.AFTER,
Container.NULL);
    Edge nextFaceEdge = _tripartite.insertEdge
(newEvert, bottom, Order.BEFORE, fvert, prevFaceEdge, Order.AFTER,
Container.NULL);
  } else {
    Edge prevFaceEdge = _tripartite.insertEdge
(newEvert, bottom, Order.BEFORE, fvert, faceEdge, Order.BEFORE,
Container.NULL);
    Edge nextFaceEdge = _tripartite.insertEdge
(newEvert, bottom, Order.AFTER, fvert, prevFaceEdge, Order.BEFORE,
Container.NULL);"
    }"
```

—replaced—

```
 "Edge prevFaceEdge = _tripartite.insertEdge
(newEvert, bottom, Order.AFTER, fvert, faceEdge, Order.BEFORE,
Container.NULL);
  Edge nextFaceEdge = _tripartite.insertEdge
(newEvert, bottom, Order.BEFORE, fvert, prevFaceEdge, Order.BEFORE,
        Container.NULL);"
```

Reason for Change: Since the structure of the internal TripartiteEPG is different in the modified version, necessary changes are also made here.

In Method "insertEdge(Vertex u, Edge eu, Order orderu, Vertex v, Edge ev, Order orderv, Object info)":

Line

```
 "Edge fe1 = _tripartite.prevIncidentEdge(e1vert, e1v);"
```

—replaced—

```
 "Edge fe1 = _tripartite.nextIncidentEdge(e1vert, e1v);"
```

Line

```
 "Edge fe2 = _tripartite.nextIncidentEdge(e2vert, e2v);"
```

—replaced—

```
 "Edge fe2 = _tripartite.prevIncidentEdge(e2vert, e2v);"
```

Reason for Change: Since the structure of the internal TripartiteEPG is different in the modified version, necessary changes are also made here.

In Method "splitVertex(Vertex v, Edge e1, Edge e2, Object info)":

Line

```
 "Edge newFaceEdge1 = _tripartite.insertEdge(newEvert, topEdge, Order.AFTER,"
```

—replaced—

```
 "Edge newFaceEdge1 = _tripartite.insertEdge(newEvert, topEdge, Order.BEFORE,"
```

Line

```
"Edge newFaceEdge2 = _tripartite.insertEdge(newEvert, topEdge, Order.BEFORE,"
```

—replaced—

```
"Edge newFaceEdge2 = _tripartite.insertEdge(newEvert, topEdge, Order.AFTER,"
```

Reason for Change: Since the structure of the internal TripartiteEPG is different in the modified version, necessary changes are also made here.

`In Methods "rightFace(Edge e)" and "leftFace(Edge e)":`

Line

```
"public Face rightFace(Edge e) {"
```

—replaced—

```
"public Face leftFace(Edge e) {"
```

Line

```
"public Face leftFace(Edge e) {"
```

—replaced—

```
"public Face rightFace(Edge e) {"
```

Reason for Change: The determination of right and left faces has to be switched according to our modification to the structure of the TripartiteEPG.

4 A Biconnectivity Augmentation Algorithm Using Tripartite Representation of EPG

Let $G - \{v\}$ denote the graph after deleting vertex v with all its incident edges from a graph G. If $G - \{v\}$ is disconnected, then v is called a cutvertex of G. If G is connected and contains no cutvertices, then G is called biconnected or 2-connected.

Here we give a biconnectivity augmentation algorithm for an embedded planar graph that takes advantage of the tripartite representation of the graph. The algorithm can run in linear time, and it is approximately optimal.

We first find all the cut-vertices incident on a face according to the rule that whether it appears more than once on the face. If a vertex appears more than once on a face, then it is a cut-vertex.

Then we proceed by condensing each block into one vertex (block-vertex). And we remove the cut-vertices with degree 2 because they are of no interest in the algorithm.

Finally we can connect the block-vertices in such a way that the augmentation is approximately optimal.

The algorithm is used in the biconnectivity augmentation of the main testing graph in the tester. The algorithm finds an optimal biconnectivity augmentation for that graph.

5 Concluding Remarks

Error checking is an important to enhance the reliability of a geometric computing library. Through testing the TripartiteEmbeddedPlanarGraph implementation of Embedded Planar Graph, we revise a few crucial mistakes and the revised version becomes more reliable.

References

[1] D. Jackson, *The TripartiteEmbeddedPlanarGraph*, private communication, 1997.

[2] G. Kant, *Algorithms for Drawing Planar Graphs*, 1993.

[3] F. P. Preparata and M. I. Shamos, *Computational Geometry, An Introduction*, Springer-Verlag, New York, 1985.