

Jean-Daniel Boissonnat Frédéric Cazals Frank Da Olivier Devillers Sylvain Pion
François Rebufat Monique Teillaud
Mariette Yvinec

INRIA Sophia-Antipolis, Project PRISME
2004 route des Lucioles, BP 93, F-06902 Sophia-Antipolis
email:firstName.LastName@sophia.inria.fr

1 CGAL: an overview

CGAL, the *Computational Geometry Algorithms Library*¹, is a large scaled project funded by the European Community. Its goal is to develop a body of objects and operations commonly used in computational geometry and to make them available to application areas and non expert users.

The CGAL project aims at providing robust, flexible and efficient code. Robustness, flexibility and efficiency are achieved through the use of state-of-the-art object oriented and generic programming techniques such as templates and trait classes. These methods are largely inspired from the STL library.

The library is organized into three parts: the *kernel*, the *basic library*, and the *support library*. The *kernel* implements basic geometric objects of fixed size such as points, line segments, lines and circles, together with some common predicates on those objects such as `orientation()` or `in_circle()` tests, intersection detection, etc. The *basic library* contains a large collection of geometric data structures and algorithms such as polygons, triangulations, planar maps, polyhedra, convex hulls, minimum enclosing circles and ellipsis, as well as trees such as kd-trees or range trees. The *support library* takes care of the communication between CGAL and visualization systems such as GeomView, OpenInventor, LEDA Window or OpenGL. Some of them are particularly well suited to the visualization of 3D objects.

The library is about 100,000 lines of C++ and can linked with other libraries written in C++, C or Fortran. Examples of such libraries are GMP number types or LEDA.

CGAL is a powerful tool for the development of industrial applications since it provides robust implementations of ubiquitous geometric structures —Delaunay triangulations and Voronoi diagrams to name two of them. Special care has been taken to handle degenerate geometric configurations —known to arise frequently in practical settings. It

¹available at <http://www.cs.uu.nl/CGAL>. Research partially supported by the Esprit program #28155 (GALIA).

should be emphasized that robustness does not preclude efficiency since the predicates the geometric algorithms rely on state-of-the-art arithmetic tools. Didactical issues have also been paid a special attention. In particular, the library can be used to get a better understanding of how geometric algorithms work by displaying the modifications of geometric data structures upon insertion/removal/update of inputs.

The goal of this video is twofold. First, it illustrates with triangulations how CGAL can be used and tuned to the users' needs. Second, it presents a convenient tool — using a 3D viewer and a multi-threaded design— to debug geometric code.

2 Programming with CGAL

In CGAL, flexibility and robustness stem from the use of object oriented techniques as well as from the available arithmetic tools. To give a better understanding of CGAL features in terms of flexibility and advanced generic programming techniques, we show how a 2D triangulation can be instantiated.

As opposed to the DCEL data structure, the representation of triangulations in CGAL is based on vertices and faces and involves three layers: the *base* classes define vertices and faces together with the incidence and adjacency relations between them, the *triangulation data structure* class is a container for faces and vertices providing the combinatorial aspects of the triangulation, the *triangulation* class implements the geometrical aspects and the user interface with the triangulation. The triangulation classes are templated with a *geometric traits* class supplying the geometric objects (points, segments, triangles...) together with the needed predicates on those objects.

Several classes of triangulations have been derived from the basic class *Triangulation_2*<*Gt,Tds*>. The most important are Delaunay triangulations, regular triangulations, constrained and Delaunay constrained triangulations. Applications of these classes are illustrated on the video. In particular, Delaunay and regular triangulations are used in shape reconstruction algorithms. Constrained and Delaunay constrained triangulations are used to mesh polygonal domains.

3.1 Goals and difficulties

Developing geometric code is known to be a difficult task: the data structures manipulated are often intricate, they sometimes involve luxuriant sets of pointers, the interactions between inputs can be elaborate, etc. This can be daunting when it comes to debugging since usual debuggers provide elementary printing facilities while one would like to be able to visualize the corresponding data structures with a 3D viewer.

A first solution consists of running two processes in parallel: the debugger and the viewer. The visualization therefore consists of (i)inserting in the source code a command such as `dumpEntityToFile(anEntity)` (ii)having the viewer scan the corresponding file(s). But this is not very convenient since the source code has to be modified and recompiled anytime the dumping instruction is inserted.

Another solution would consist of invoking a command such as `dbx>call prettyPrintEntity(anEntity)` from the debugger. (Remind that the `call` command provided by `dbx` or `gdb` enables the user to execute a function interactively.) This would be a 3D version of `dbx>print aVariable`. We call this debugging mode *non intrusive debugging* since the source code is not modified. Since the items to be displayed have to be added to the viewer, the viewer's resources must be visible from the geometric code under debug. And since two different processes do not have the same address space and cannot see one another variables, the geometric code and the viewer cannot be run in different processes.

As a conclusion, non intrusive debugging requires running “at the same time” the debugger and the visualizer with the constraints that their variables and resources are mutually visible. We show that multi-threading provides a simple, portable and efficient solution.

3.2 An overview of threads

The following quote from [Mue92] lists the main features of threads and explains why multi-threading offers a convenient way to perform easily complementary tasks:

A thread is an independent flow of control within a process. Threads support the concept of concurrent programming: They may be created and terminated dynamically, such that multiple threads may exist within one process. Threads within the same process share global data (global variables, files, etc) but maintain their own stack, local variables, and program counter. Threads are referred to as light-weight because their context is smaller than the context of processes. ... They also provide a simple but powerful model for exploiting parallelism in a shared-memory multiprocessor environment. The POSIX threads extension specifies a priority-driven thread model with preemptive scheduling policies, signal handling, and synchronization primitives to allow mutual exclusion.

An important characteristic of multi-threaded programs is to have shared data between multiple threads. A first consequence is to impose sequential access to these resources. Another consequence is to require sets of instructions to be executed in sequence without being sliced by the scheduling. For example, a thread filling up an array and updating the array index should complete its job before the array is parsed—or the last item may be missed. The tools available

to synchronize threads are *mutexes* and *conditions variables*.

Mutexes (from *mutual* and *exclusion*) are the basic primitive. Basically, a thread uses a mutex by locking it, performing some job on shared data, and unlocking it. If the mutex is already locked, the calling thread is put in a waiting queue.

Condition variables are a signaling mechanism used to convey information about shared resources and activate the threads accordingly. See [But97] for more informations.

3.3 Debugging strategy

From the discussion of section 3.1, we aim at providing a 3D equivalent to `dbx>print aVariable` for geometric entities. The method we propose uses two threads: a thread running the geometric code to be debugged, and a thread running an Open Inventor viewer—see fig. 1. The program `dbx` is run on is a `main()` wrapper launching these two threads.

When the geometric thread is active, it can be examined as usual. In particular, the programmer can stop anywhere using a breakpoint. Consider the situation where the programmer is stopped and wants to visualize a (set of) geometric item(s) *without* going any farther in the geometric code. This requires: (i)adding the items to be visualized to Open Inventor's scene graph (ii)interrupting the geometric thread right away to give the flow control to the visualization thread (iii)being able to stop the visualization thread and wake up the geometric one after the visualization has been performed.

This can be implemented using mutexes and condition variables. See [Caz99] for the details.

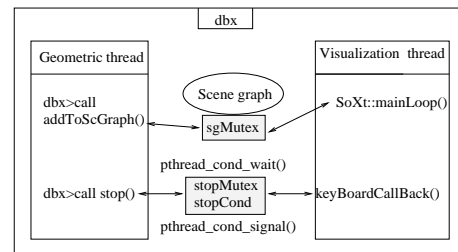


Figure 1: Geometric and visualization threads under the debugger

A similar strategy can be used to incrementally visualize the updates undergone by geometric data structures in a demonstration context.

References

- [But97] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
- [Caz99] F. Cazals. Non-intrusive debugging and incremental visualization of geometric code. *Submitted*, 1999.
- [Mue92] F. Mueller. Implementing posix threads under unix: description of work in progress. In *Proc. of the 2nd software engineering research forum*, Melbourne, Florida, 1992.