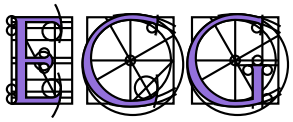
ECG

IST-2000-26473

Effective Computational Geometry for Curves and Surfaces



ECG Technical Report No. : *ECG-TR-361215-01*

Testbed implementations of exact and approximate algorithms

Efraim Fogel

Astrid Sturm

Deliverable: 36 12 15 (item 01)
Sites: TAU, FUB
Month: 36

Project funded by the European Community
under the "Information Society Technologies"
Programme (1998–2002)

Testbed implementations of exact and approximate algorithms*

Efraim Fogel†, Astrid Sturm‡

April , 2004

1 Introduction

This document describes a practical toolkit to evaluate the status of code. It can be used to create programs that measure performance, known as benchmarks, and other various tests, execute them, and analyze their results. With little efort a user of this toolkit can detect inefciencies, bottlenecks, and loss of functionality or performance degradation, compare various techniques, algorithms, and diferent implementations, and measure progress. A user, can then, present the results in a comprehensible way. The information produced also includes the precise description of the execution environment to allow the reproduction of the results.

The toolkit consists of three parts that corresponds to the three phases required to evaluate code; (1) The creation of a hierarchy of test or benchmark programs, (2) the selective and controlled execution of the programs with various input test cases, and (3) the analysis and proling of their results and the conversion of the data into more meaningful and comprehensible presentations. The following Sections describe the three parts in details.

It must be noted that our goal here is not to replace existing tools that already provide useful functionality for computational experiments (e.g., gnuplot, make, perl, python). Rather, the goal is to augment this set with new tools that build on the functionality already available to provide a comfortable testingenvironment.

It's worth mentioning the ExpLab tool set for Computational Experiments by Susan Hert, Lutz Kettner, Tobias Polzin, and Guido Schäfer from the Max-Planck-Institut für Informatik. There seems to be little overlap between ExpLab and our toolkit, but for most they emphasize diferent aspects.

The code is available from the web page
`http://www.inf.fu-berlin.de/~rote/Software/Software-for-approximation-of-curves.html`

*Partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces)

†Tel Aviv University,School of Computer Science, Schreiber building, Tel Aviv 69978, Israel. efif@post.tau.ac.il

‡Freie Universität Berlin, Institut für Informatik, Takustraße 9, D-14195 Berlin, Germany. sturm@inf.fu-berlin.de

2 Creation

The toolkit was developed as a package in CGAL, the Computational Geometry Algorithms and Data Structures library. As a CGAL module on its own, it obliges to the Generic Programming Paradigm. EFI SAYS: (Add space after CGAL.) The first part consists of a couple of generic C++ classes, and the ← interface between them and the user code to be evaluated.

The `Bench_parse_args` class is used to parse command line options, interpret them, and configure the bench accordingly. The bench itself is performed through the () operator of the `Bench` class described next.

The Bench class is parameterized with a model of the Benchable concept. This concept serves as the interface between the measuring device and the operation you wish to measure, refereed as the target operation here after. A model of this concept must satisfy a few requirements as follows. It must have a default constructor, and it must provide four methods. The `init()` method can be used to initialize some data members before the target operation is carried out. The `clean()` method is used to clean those data members and residual data of the operation after the measure is completed. The `sync()` method can be used to synchronize between the target operation and the time-sampling operations. While these three methods must be provided, they can be empty. The `op()` method performs the target operation. This operation is executed in a controlled loop according to various criteria explained in the next paragraph.

If the estimated time-duration it takes to complete a single execution of the target operation is large compared to the granularity of the timing device, it is suficient to execute the target operation a small number of times between sampling the timing device, perhaps even once. You can set the number of times the target operation is executed through the `set_samples()` interface method of the Bench class. On the other hand, if the estimated time-duration of a single execution is of the same order as the granularity of the timing device, increasing the number of executions increases the accuracy of the measure. With the `set_seconds()` method of the Bench class, you can set the time slot in seconds allocated for the measure. In this case the target operation is executed in a loop while the number of executions is counted. When the time expires, the counter is sampled. Then, the target operation is executed again for as many times as counted, while measuring the time it takes to complete the sequence.

2.1 A simple Benchmark Program

The basic structure of a useful benchmark program can be very simple; Its tasks are to measure the time it takes to complete a sequence of executions of the target operation, and present the results in some useful manner.

Figure 2 contains a listing of a program that produces the results shown in figure 1

The () operator of the **Bench** class counts the number of times the function **sqrt()** can be executed within the allocated time slot. Then, it executes the **sqrt()** function for as many times as counted, while measuring the time it takes to complete the sequence.

```
Bench                            Bench    Ops      Total    Single   Num Ops
    Name                             Time      Num Ops Time  Op Time  Per Sec
-------------------------------- -------- -------- -------- -------- --------
Square root                             1 14690512   1.0000   0.0000 14690512.0000
```

Figure 1: Square-root benchmark results.

By default the allocated time slot is 1 second. The program overrides the default with the number of seconds returned by the **get_seconds**() method of the **Bench_parse_args** class. The default of the later is 1 second as well, and can be overridden with the "**-t** ***seconds***" command-line option.

3 Execution

This section lists the various command-line options a benchmark program may accepts, and it explains how to create a hierarchy of programs and execute the programs in the hierarchy selectively using an agent implemented in **Perl**.

Some pieces of the toolkits are dedicated to the development and maintenance of the CGAL planar map modules. For example, some of the command-line options listed below directly control the behavior of the planar-map benchmarks, other users may ignore them for the time being.

3.1 Command-Line Options

A program written with the aid of the **Bench_parse_args** class accepts the command-line options listed below. The command-line options must be provided after the executable name and before an optional name of an input file. A brief description is displayed on the console as a response to the " **-h**" option.

-b ***options*** set bench options

type_name=*type*

tn=*type* set bench type to *type* (default all).
type is one of:

i[ncrement]	**0x1**
a[ggregate]	**0x2**
d[isplay]	**0x4**

type_mask=*mask*

tm=*mask* set bench type mask to *mask*

strategy_name=*strategy*

sn=*strategy* set bench strategy to *strategy* (default all).
strategy is one of:

t[rapezoidal]	**0x1**
n[aive]	**0x2**
w[alk]	**0x4**
d[ummy]	**0x8**

```
#include <math.h>
#include <CGAL/Bench.h>
#include <CGAL/Bench.C>
#include <CGAL/Bench_parse_args.h>
#include <CGAL/Bench_parse_args.C>

class Bench_sqrt {
private:
  double n;
public:
  Bench_sqrt() : n(M_PI) {}
  int init(void) { return 0; }
  void clean(void) {}
  void sync(void) {}
  void op(void) { sqrt(n); }
};

int main(int argc, char * argv[])
{
  CGAL::Bench_parse_args parseArgs(argc, argv);
  int rc = parseArgs.parse();
  if (rc > 0) return 0;
  if (rc < 0) return rc;
  CGAL::Bench<Bench_sqrt> bench("Square root", parseArgs.get_seconds());
  bench();
  return 0;
}
```

Figure 2: Square-root benchmark

	strategy_mask=*mask*	
	sm=*mask*	set bench strategy mask to *mask*
	h[eader]=*bool*	print header (default **true**)
	name_length=*length*	
	nl=*length*	set the length of the name field to *length*
-d *dir*	add directory *dir* to list of search directories	
-h	print this help message	
-i *iters*	set number of iterations to *iters* (default 0)	
-I *options*	set input options	
	f[ormat]=*format*	set format to *format* (default **rat**).
		format is one of:
		i[nt] integer
		f[lt] floating point
		r[at] rational
-r *root*	set the `$ROOT` to *root* (default is the environment variable `$ROOT`)	
-s *samples*	set number of samples to *samples* (default 10)	
-t *seconds*	set number of seconds to *seconds* (default 1)	
-v	toggle verbosity (default **false**)	

3.1.1 Input

The sole input file, if provided, must appear after the last command-line option in the command line. This file is searched for in a directory search-list. The initial list consists of the current directory followed by `$ROOT/data/Segments_2`, `$ROOT/data/Conics_2`, and `$ROOT/data/Polylines_2` in this order, where `$ROOT` is initialized with the value of the environment variable `$ROOT`, and possibly overridden using the command line **"-r *root*"**

The **"-d *dir*"** command-line option inserts the directory ***dir*** at the end of the search list.

The **get_input_format**() method of the **Bench_parse_args** class returns the format provided by the user through the **"-I format=*format*"** command line option, or **"-I f=*format*"** in short.

3.1.2 Output

The output produced by a single benchmark program exemplified in figure 1 is a text-based table easily readable by humans. It consists of an optional header record and a data record. The production of the header can be suppressed by the **"-b header=false"** command-line option, or **"-b h=false"** in short. Occasionally a sequence of benchmarks are performed in a raw and the display of the header is desired only once (or once per page). The **print_results**() function, which produces the output is virtual. The user can start with the original function and change/add whatever he wants.

A original data record consists of the following fields:

Bench Name the name of the benchmark.

Bench Time the allocated time-slot for the entire benchmark in seconds (see **"-t *seconds*"** option).

Ops Num the number of target operations performed within the time slot.

Total Ops Time the time required to perform the loop of operations consisting of **Ops Num** operations in seconds.

Single Op Time the average time required to perform a single operation in seconds.

Num Ops Per Second the number of operations completed per second.

For the approximation algorithms the data record got extended by:

Output Num the number of output elements produced by the target operation. For the polygonal approximation this is the number of vertices of the approximating polygonal curve.

The **Bench Name** field identifies the benchmark for all purposes, its length is 32 characters by default. The length can overridden by the **"-b name_length=*length*"** command-line option, or **"-b nl=*length*"** in short.

Independent tools listed in section 4 parse log files that contain benchmark results, manipulate them, analyze them, and perhaps convert them to other formats for artful presentations.

3.2 Execution of multiple benchmarks

The **Perl** script **cgal_bench** selectively executes multiple benchmarks ordered in a hierarchy. It executes them in a sequence one at a time passing the appropriate command-line options and input data file for each execution. It accepts a few command-line options on its own listed below, and reads an input file that contains the hierarchy of the benchmarks along with necessary data required to execute them. This information is represented in a simple language derived from the Extensible Markup Language (XML).

3.2.1 Command-line options

-args *args*	EFI SAYS: (Remove space before -args.) set additional arguments passed to the benchmark programs. ←
-help	print this help message.
-verbose *level*	set verbose level to ***level*** (default 0).
-database *file*	set database xml file to ***file*** (default `$ROOT/bench/data/benchDb.xml`).
-filter *name*	select bench ***name***, and sub benches (default all).

A unique prefix is sufficient to indicate the desired option. For example, when the **"-help"** option is specified, a brief description is displayed on the console, and the script quits immediately after. The same behavior is achieved through the abbreviated **"-hel"**, **"-he"**, and **"-h"** options.

By default the script reads the file `$ROOT/bench/data/benchDb.xml`. This can be overridden through the **"-database *file*"** command-line option.

3.2.2 Documenting the Environment

The scripts automatically documents the environment in which it performs the benchmarks, so the benchmark can be easily rerun (provided the same environment is still available) and the results can be more accurately compared to the results of other benchmarks.

The scripts extract most of the information directly from the environment. Additional configuration data that cannot be extracted directly from the environment is extracted from the database input file. The script prints out its finding, and only then it starts performing the benchmarks. Here is an excerpt from a sample run of `cgal_bench`, showing the type of information extracted and printed out.

```
Mon Mar 31 20:29:02 2003
COMPILER NAME: gcc
COMPILER INFO: gcc (GCC) 3.2
Copyright (C) 2002 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

OS NAME: linux
OS INFO: Linux cgal 2.4.20-net1 #1 SMP Wed Feb 5 13:05:52 IST 2003 i686 unknown

PROCESSOR: 0
CPU SPEED: 999.783 MHz
CPU TYPE: Pentium III (Coppermine)
PRIMARY DATA CACHE: 256 KB
SECONDARY DATA CACHE: 0
INSTRUCTION CACHE: 0
PROCESSOR: 1
CPU SPEED: 999.783 MHz
CPU TYPE: Pentium III (Coppermine)
PRIMARY DATA CACHE: 256 KB
SECONDARY DATA CACHE: 0
INSTRUCTION CACHE: 0
MEM SIZE: 2020 MBytes
GFX BOARD: unknown
CGAL VERSION: 2.5-I-81
LEDA VERSION: 441
QT VERSION: unknown
```

3.2.3 Input File Format

The representation of the input file is derived from XML. Its element tag-set consists of 4 predefined element tags listed below. All other element tags that appear in an input file without exception are names of executables that perform benchmarks.

The following is a list of the 4 elements with the 4 predefined tags respectively:

file specifies a file.

bench specifies a hierarchy of benchmarks.

clo specifies a command-line option.

class specifies a style-sheet class.

A **file** element specifies a data file provided as input to a benchmark. It may have the following attributes:

name - the file name.

format - the number type.

curves - the number of curves.

vertices - the number of vertices.

halfedges - the number of halfedges.

faces - the number of faces.

The **name** attribute is mandatory, as it identifies the file for all purposes. The other attributes are optional (as a matter of fact, the last 4 attributes are specific to the planar-map benchmarks.)

A **clo** element specifies a command-line option. It has the following two mandatory attributes:

name - the option name.

string - the option string.

The option name identifies the option for all purposes. The option string is the exact argument that must appear in the command line for that option to take effect.

A **bench** element specifies a hierarchy of benchmarks. It can contain multiple **file**, **clo**, or **class** elements, multiple elements that represent executables, and multiple nested **bench** elements. The **file** attribute of a **bench** element, if present, specifies an input data file. The value of the **file** attribute is the name of the input file (and the value of the **name** attributes of a **file** element). Each attribute of a **bench** element that is neither **file** nor **enable** specifies a command-line option. The value of such an attribute is the option variable-value or parameter. Command-line options are passed through inheritance in the benchmark hierarchy, where an option parameter specified in a **bench** element overrides the parameter specified higher in the hierarchy. The boolean attribute **enable** simply indicates whether the bench element is enabled or disabled. Disabling a bench elemnt disables all it descendants and essentially prunes the hierarchy tree. EFI SAYS: (Changed the wordings of the last sentence.) ←

The tag of a benchmark element is the name of an executable that performs a benchmark. A benchmark element, just like a **bench** element, may contain a **file** attribute to indicate an input data file, an EFI SAYS: (Changed a to an) **enable** attribute to enable or disable the benchmark, and multiple attributes, ← each indicating the parameter of a command-line option.

A **class** element is used only while generating files for browsing (e.g., html, php, etc.).

Figure 3 lists a simple bench input-file that consists of three benchmarks, three corresponding input files, and some command-line options that are used to execute the benchmarks. When this file is provided as input to the **cgal_bench** script, the later parses the file, interprets its contents, and executes the commands below in turn:

```
bench1 -s 10 -bh=true -bnl=64 file1
bench2 -s 10 -bh=flase -bnl=64 file2
bench3 -s 10 -bh=flase -bnl=64 file3
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bench name_length="64" header="false">
  <clo name="samples" string="-s"/>
  <clo name="name_length" string="-bnl="/>
  <clo name="header" string="-bh="/>
  <clo name="format" string="-lf="/>

  <file name="file1" format="rat"/>
  <file name="file2" format="rat"/>
  <file name="file3" format="rat"/>

  <bench samples="10" enable="true">
    <bench1 file="file1" header="true"/>
    <bench2 file="file2"/>
    <bench2 file="file3"/>
  </bench>
</bench>
```

Figure 3: A simple database

4 Analysis

The scripts in this category are intended to analyse and profile results of benchmarks and convert the resulting data into more meaningful and comprehensible presentations. They parse log files that contain benchmark results, interpret the data, manipulate it, analyze it, and perhaps convert it into other formats for artful presentations.

In principle they should be able to 1. add new fields to the predefined fields of a benchmark data-record. This can be useful to perform mathematical operations on the data values, or to reformat the output

for pretty printing, 2. merge records, 3. sort the records, 4. filter out records, and 5. convert the data into other formats.

As different applications have varying requirements for the result data presentation, converting the textual results into other formats that support artful presentations, pretty printing, and browsing capabilities is left to the user. The **bash** command-line below can be used to record benchmark results in separate log files in the `~/logs` directory:

```
cgal_bench <flags> 2>&1 | tee ~/logs/bcgal_‘date +%y%m%d%%%H%M%S‘.log
```

When these log files are sorted by name, which is typically the default, it is fairly easy to point at the most recent log file.

5 Reference Guide

Concept Benchanble

Creation

Benchanble benchable;

A default constructor.

Operations

void	*benchable.init(void)*	this function is invoked once before the function that tests the target operation is invoked. It is used to initialize all the data members that are operands consumed by the target operation. This allows for accurate measure of the target-operation performance, through timing of the function that performs the target operation.

EFI SAYS: (Added 'the' before target operation....) ←

void	*benchable.clean(void)*	this function is invoked once after the function that tests the target operation returns. It is used to clean up the data members in general and the results in particular produced by the target operation.
void	*benchable.op(void)*	this function performs the target operation repeatedly in loop constrained, either by a maximum number of repetitions, or by by a limit on the time period alloted for the target operation, depending on the bench configuration.

void	*benchable.sync(void)*	this function synchronizes between the target operation and the measure of its time consumption. It is called once after initialization (before the starting time is sampled), and once again after the target operation is performed in a loop and before the ending time is sampled. In many cases the *sync* function is empty, as there is no need to synchronize anything. On the other hand, suppose that the performance of a sequence of commands in pipeline architecture is to be measured. In this case the *sync* function can be used to flush the pipeline and force all pending commands to complete before the ending time is sampled. This situation manifests itself when the performance of some **o**penGl (see cite) commands is to be measured. The *sync* function can be implemented to call **g**lFlush to cause all issued commands trapped in intermediate buffers of the pipeline to be executed as quickly as they are accepted by the actual rendering engine before they are assumed to complete.

EFI SAYS: Removed 'and' and added parentheses.) ←—

Class Bench

Definition

Bench is a utility class that performs the bench. The benchmark is configured during construction of the *Bench* class or through some modifiers.

#include <*CGAL/Bench.h*>

Creation

Bench bench(std::string name = "", int seconds = 1, bool printHeader = true);

constructs an instance of the *Bench* class.

Modifiers

void *bench.set_iterations(int iterations)*

void *bench.set_seconds(int seconds)*

void *bench.set_samples(int samples)*

Query Functions

int *bench.get_iterations()*

int *bench.get_seconds()*

int *bench.get_samples()*

Benchable& *bench.get_bench_user()*

virtual void *bench.print_results()*

void *bench()*

Class Bench_parse_args

Definition

#include <*CGAL/Bench_parse_args.h*>

Enumerations

Format_id

Type_id

Strategy_id

Creation

Bench_parse_args *bench_parse_args(int argc, char * argv[]);*

constructs an instance of the *Bench_parse_args* class.

Query Functions

unsigned int *bench_parse_args.get_Type_mask()*

unsigned int *bench_parse_args.get_Strategy_mask()*

bool *bench_parse_args.get_verbose()*

Format_id *bench_parse_args.get_input_format()*

int *bench_parse_args.get_samples()*

int *bench_parse_args.get_iterations()*

int *bench_parse_args.get_seconds()*

bool *bench_parse_args.get_print_header()*

int *bench_parse_args.get_name_length()*

*const char** *bench_parse_args.get_filename()*

*const char** *bench_parse_args.get_fullname()*

*const char** *bench_parse_args.get_type_name(Type_id id)*

*const char** *bench_parse_args.get_strategy_name(Strategy_id id)*

EFI SAYS: Added Enumerations and changed FormatId, TypeId, and StrategyId to Format_id, Type_id, and Strategy_id ←