BlackBox Component Builder

For Scientists and Engineers

Scientific programming without a sting.

To be presented at LLE in February 2004

Wojtek Skulski

Laboratory for Laser Energetics
University of Rochester

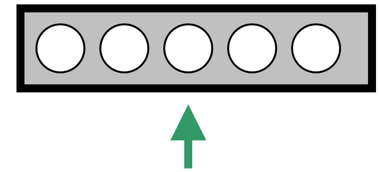
Rochester New York 14623-1299

skulski _at_ pas . rochester . edu

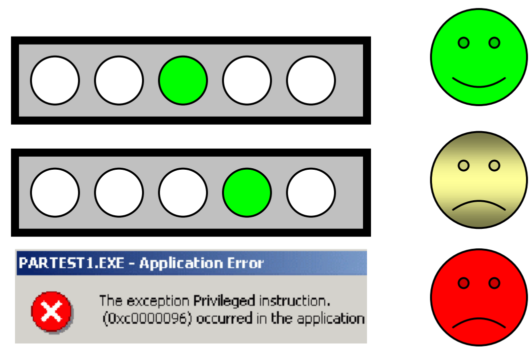
Main points of this presentation

- Why BlackBox / Component Pascal ?
- How is a BlackBox application different from a classic "program”?
- How to establish a good structure from start?
- I have a DLL. How do I interface it with BB?
- How to make a powerful GUI without ever bothering about one?
- How to use waveform graphics BlackBox/Gr?
- How to use the Graphics and Scientific Library ***Lib*** (graphics, matrices, vectors, special functions, digital filters, etc.)
- What is available from the web: graphics, utilities, etc.
- Which textbooks are good to read?

A common problem



- I want to achieve some goal, e.g., turn on an LED.
- I write a piece of software and push the GUI button.
- What can happen next:

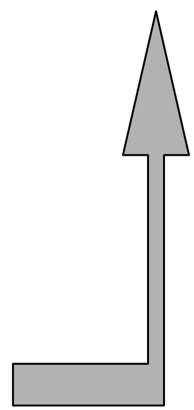


1. The intended LED lights up. Hurrah!
2. Another LED lights up. Oops.
3. Internal application error?!

   The application disappears from the screen.

Ad. 2. I should correct my algorithmic mistakes.

Ad. 3. This should never happen! But it often does.



Let it never happen again to us!

Is it acceptable to crash?

A catastrophic failure ("crash") may or may not be acceptable.

1. A grad student wrote an analysis program that dumps core on wrong data.

   → Acceptable, if it does not delay his thesis (and his adviser's papers).

2. The student's SW is used to collect data. The DAQ is crashing every hour.

   → Annoying, but acceptable, if collected data is still adequate to write papers.

3. Navigation SW crashed during descent. The spacecraft smashed onto the Moon.

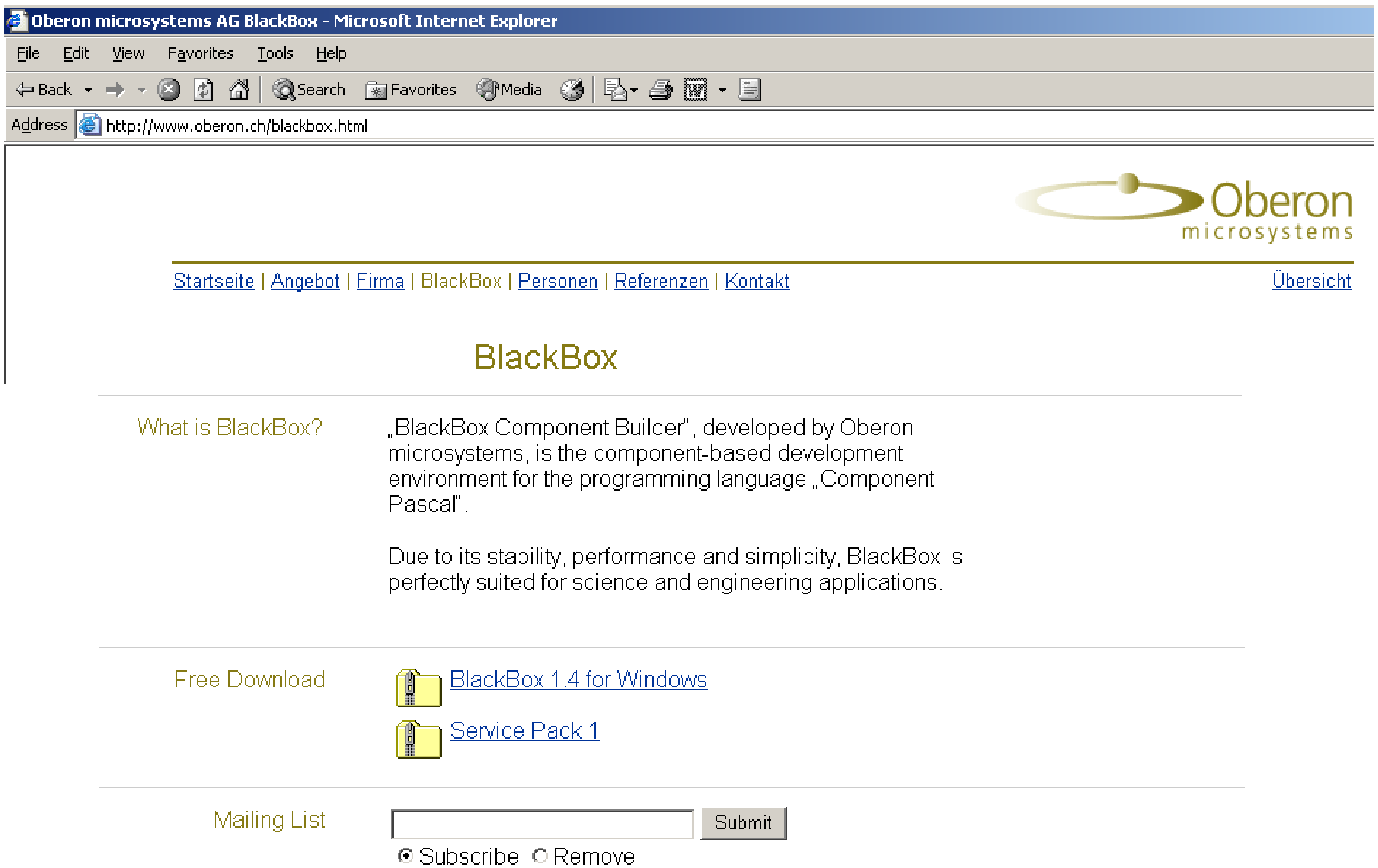
   → The entire lunar mission failed, hundreds M$ lost.

Programming techniques that are acceptable for either #1 or #2, are not acceptable if the situation resembles #3.

The solution: BlackBox Component Builder (free!)

Download BlackBox from the website **http://www.oberon.ch/blackbox.html**

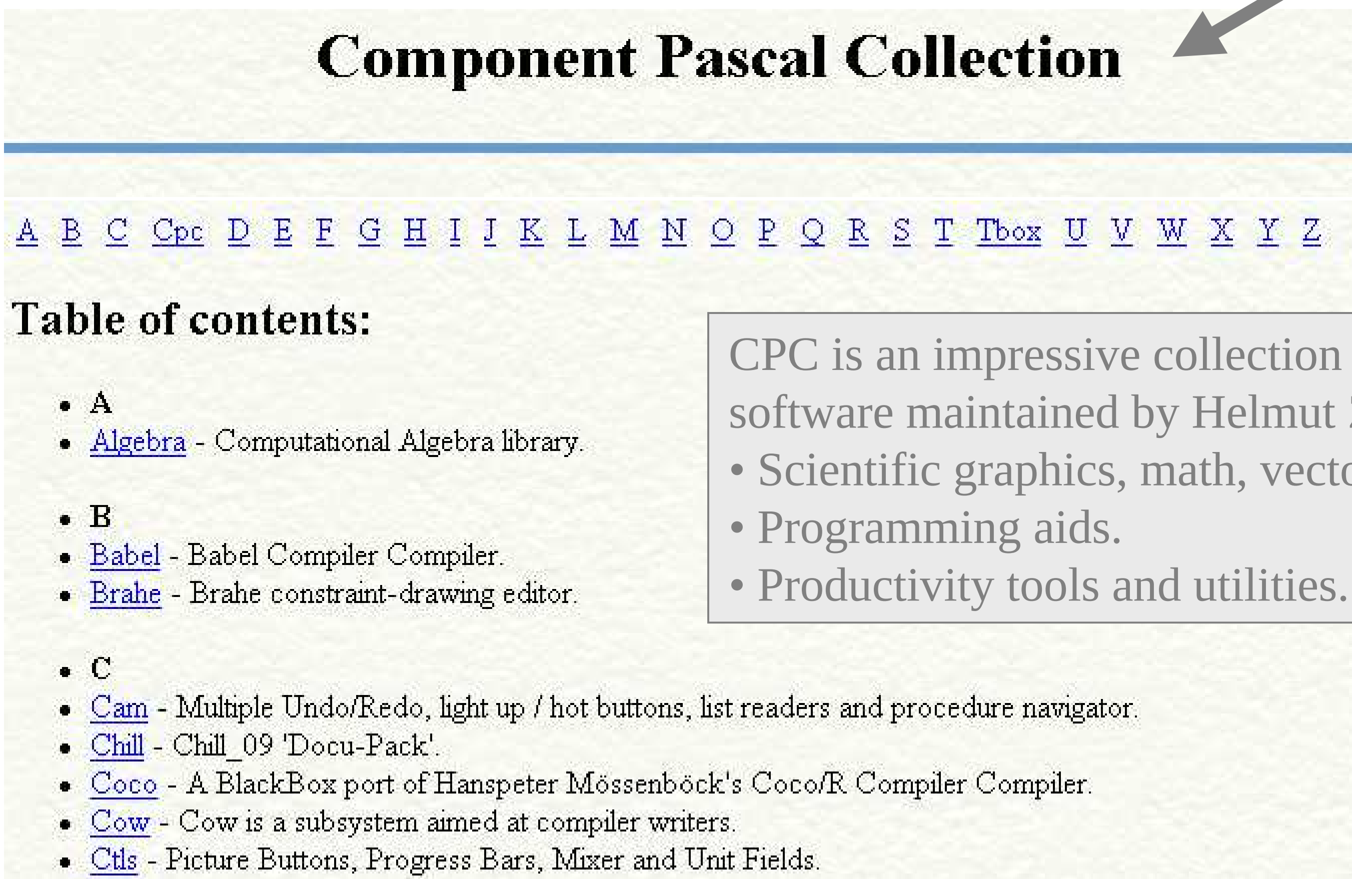
After downloading and unzipping the files, follow the installation instructions.



BlackBox User Community: scientists, engineers, programmers

Subscribe to BlackBox Users mailing list BlackBox _at_ oberon.ch (replace _at_ with @)

Choose BlackBox software from Component Pascal Collection **http://www.zinnamturm.de/**



Component Pascal Collection

A B C Cpc D E F G H I J K L M N O P Q R S T Tbox U V W X Y Z

Table of contents:

- **A**
- Algebra - Computational Algebra library.

- **B**
- Babel - Babel Compiler Compiler.
- Brahe - Brahe constraint-drawing editor.

- **C**
- Cam - Multiple Undo/Redo, light up / hot buttons, list readers and procedure navigator.
- Chill - Chill_09 'Docu-Pack'.
- Coco - A BlackBox port of Hanspeter Mössenböck's Coco/R Compiler Compiler.
- Cow - Cow is a subsystem aimed at compiler writers.
- Ctls - Picture Buttons, Progress Bars, Mixer and Unit Fields.

CPC is an impressive collection of BlackBox software maintained by Helmut Zinn.
• Scientific graphics, math, vectors/matrices.
• Programming aids.
• Productivity tools and utilities.

One way to get started with BlackBox: by example

1. Download and install BlackBox from **http://www.oberon.ch/blackbox.html**

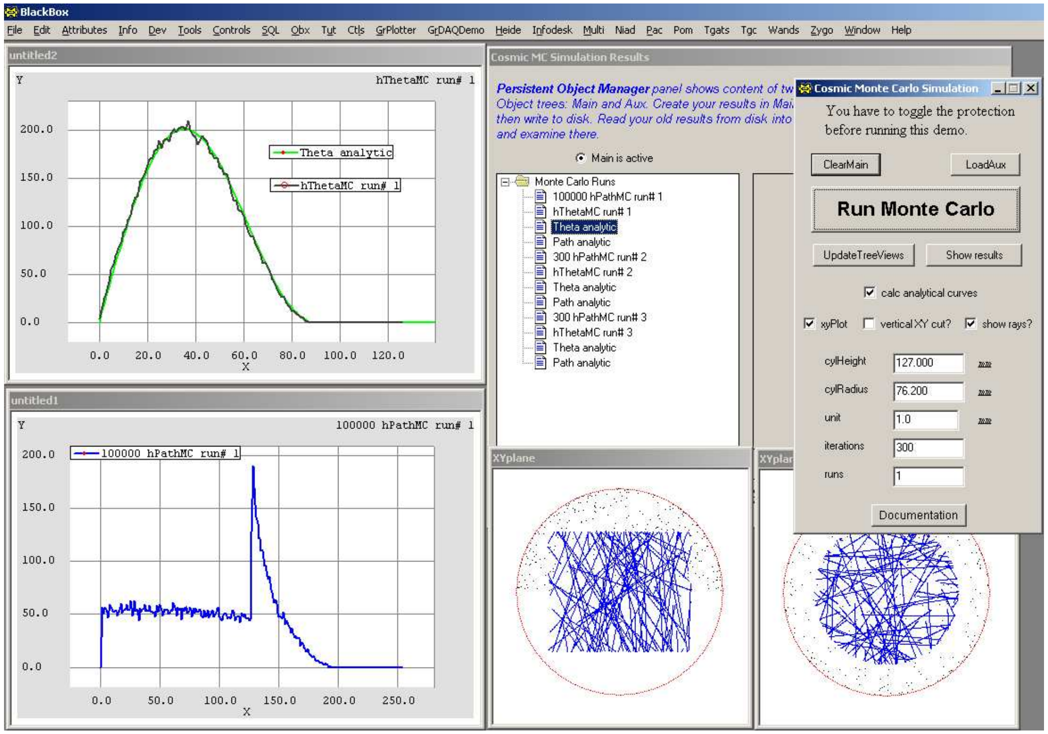
   Hint: for convenience use directory C:\Blackbox for installation.

2. Download and install **Gr** and **POM** from

   http://www.pas.rochester.edu/~skulski/Downloads.html

   Installation is simple: unpack the ZIP archives into C:\BlackBox created in step 1.

3. Start BlackBox, find menu item **Pom**, and begin exploration by reading POM manuals.



Literature: Component Pascal and Oberon-2

J. S. Warford **Computing Fundamentals**
The Theory and Practice of Software Design with BlackBox Component Builder
Vieweg 2002, ISBN 3-528-05828-5, http://www.vieweg.de

N. Wirth and M. Reiser **Programming in Oberon - Steps Beyond Pascal and Modula**
Addison-Wesley, 1992, ISBN 0-201-56543-9

Excellent tutorial for the Oberon programming language and language reference.

H. Mössenböck **Object-Oriented Programming in Oberon-2**
Springer, 1995, ISBN 3-540-60062-0

Principles and applications of object-oriented programming with examples in Oberon-2.

C. Szyperski **Component Software**
Addison-Wesley 2002, 2nd edition, ISBN 0201745720

N. Wirth and J. Gutknecht **Project Oberon - The Design of an Operating System and Compiler**
Addison-Wesley, 1992, ISBN 0-201-54428-8

See also links at http://www.oberon.ethz.ch

The Method

Clear structure

Safe programming

Prerequisites to develop trustworthy software

Programs crash because of internal errors.

Internal errors can be avoided with safe programming.

Safe programming has to be enforced rather than recommended.

Communication and structure.

The goal of a program is communication with other people.

Communication needs structure.

Structure has to be enforced rather than recommended.

The only reliable policeman is the compiler.

Optional tools may not be used (because of “lack of time”).

Internal errors are avoidable

Programs crash because of internal errors.

- Array indices out of bounds → segmentation violation.
- Pointer arithmetic errors → segmentation violation.
- Dangling pointers → segmentation violation.
- Unguarded typecasts → segmentation violation.
- Memory leaks → running out of memory.

All the above can, and have been, eliminated from modern languages:

Oberon, Component Pascal, Java, and C#.

The reliable way to avoid internal errors is to use one of the above,

and to avoid Fortran, C, or C++.

Large programs are maintainable if they have structure

Programs and projects often fall apart because of uncontrollable growth of their size and deterioration of their structure.

> Even a small program can be impossible to maintain if its structure is a mess.

Structure is meant for humans who develop and maintain programs.

Hierarchical structure is human-oriented.

Books	→	subsystems.
Chapters	→	modules.
Paragraphs	→	procedures and data structures.
Sentences	→	programming statements and variables.

Good programming language should encourage hierarchical structure.

Component Pascal is both safe and supports structure

Component Pascal (C.P.) is one of few modern programming languages that eliminated sources of common internal errors.

- Array bounds are checked.
- Memory management is automatic (“garbage collection”).
- Typecasts are always guarded. (Never crash the runtime system.)
- Types checked during compilation (“strongly typed language”).

Other “safe languages” include Oberon, Java, and C#.

C.P. has been extensively used in mission-critical applications (including the one at LLE) and it does fulfill the promise.

In addition, C.P. development is efficient, code performance is known to be very good, and it directly supports hierarchical structure.

Where does it come from?

40+ years of development and refinement

Component Pascal: historical perspective

- How an electrical engineer changed the computer science.
  - Electrical engineers know that circuits should not burn or explode.
  - Computer scientists assumed that if SW explodes, they can reboot.
  - An electrical engineer said “stop the madness”. His name: Prof. Niklaus Wirth.

- 1960. Algol: designed by a committee with N.Wirth participation.
- 1968-1970. Pascal: data structures, pointers, block structure, recursion.
- 1977-1979. Modula-2: modules, interfaces, information hiding, separate compilation. Program development in teams, software engineering.
- 1986-1988. Oberon: type extensibility, inheritance, object-orientation.
- 1992. Oberon-2: type-bound procedures (methods).
- 1997. Component Pascal. Industrial implementation of Oberon-2.

BlackBox = Component Pascal + framework + operating environment.

BlackBox = over 40 years of experience and refinement.

BlackBox: historical perspective

- BlackBox is an operating environment that originated as an operating system.
  - 1985-1989. Oberon System: an operating system developed by Wirth and Gutknecht.
  - Ran on 1-MIPS, 32-bit engineering workstation also designed by Wirth *et al*.
  - Text, graphics, networking, e-mail, compiler, and run-time in 12k lines of code.
- 1989-1992. Oberon System ported to DOS, Windows, MacOS, several Unices.
  - Ported versions ran under respective OS's, but had original look-and-feel.
  - Full portability of source code between all ports.
  - *Develop anywhere, run anywhere* has been achieved and demonstrated long before Java.
  - 1992-…. Oberon System-3 for Windows, Linux, Solaris, and bare hardware (Intel).
  - This system is now known as BlueBottle. See www.oberon.ethz.ch for the latest details.
- 1992. Oberon System commercialized under the name Oberon/F (Windows, Mac).
- 1997. Oberon/F renamed to Component Pascal and BlackBox.
  - A complete package: editor, compiler, debugger, loader, and runtime kernel.
  - Hosted under Windows, with Windows look-and-feel.
  - Over the years, users developed math packages, graphics, and applications.

Predicting the future of computing

- Predictions are difficult, especially if they concern the future (Winston Churchill).
- Industry has realized that the current state of software is worrisome.
- Lack of robustness is caused by languages such as FORTRAN, C, or C++.
- Tried Java to address the problem.

  Java was modeled after Oberon.

  But Java runtime system had poor performance.
- Finally, Microsoft hired one of chief BlackBox developers to design .NET.
- .NET will incorporate lessons learned by the N.Wirth school over 40+ years.
- Transition from Windows to new technology will be as long and gradual as transition from DOS to Windows. All the foundations need be reworked.
- Eventually, C++ will be discarded (or reworked and renamed to C+#?&).
- We do not have to be at the bleeding edge. We have a solution right now.
- BlackBox robustness has been proven more than once.

Why is BlackBox good for scientific and engineering applications?

Math libraries

Graphics

Interfacing with hardware

Efficient code

Features that are needed for scientific an engineering apps

Simple yet powerful programming language.

Iron-clad runtime system.

Instantaneous compile/load/debug cycle.

Comprehensive math libraries by Robert Campbell (BAE Systems).

Comprehensive graphics.

- Scientific 1D, 2D, and 3D plotting by Robert Campbell, BAE Systems.
- Interface to OpenGL.
- Waveform graphics by Wojtek Skulski, University of Rochester.

Easy to interface with hardware (via hardware-specific DLLs).

Excellent support from the vendor.

Knowledgeable user community, quick response to questions.

Free. Source released at the end of 2004.

Let's have a look at some BlackBox applications

- We can assess the strength of the tool and the community by looking at a few examples.

☺ BUGS (**B**ayesian inference **U**sing **G**ibbs **S**ampling).

Large cooperative project led by Imperial College School of Medicine and University of Helsinki with over 15 years of development. Application area: Markov chain Monte Carlo applied to pharmacokinetic models and epidemiological studies.

Search Google for "WinBUGS" and explore the websites.

☺ Russian Academy of Sciences: Theoretical calculations for High Energy Physics.

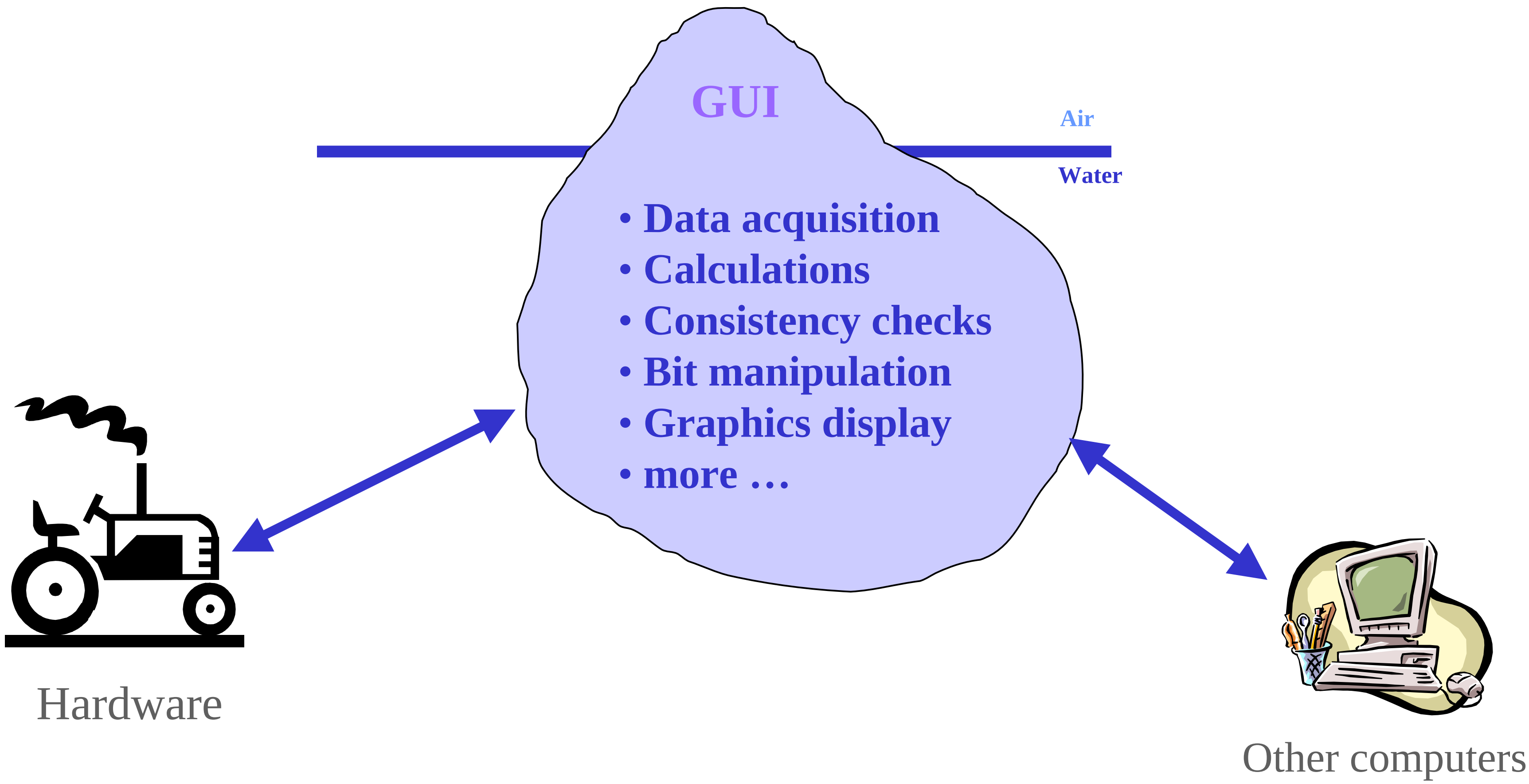
☺ BAE Systems: Antenna modeling for airborne radar systems.

☺ Oberon Microsystems: Monitoring software for a hydro power station.

Latter three described at www.cern.ch/Oberon.Day → Presentations

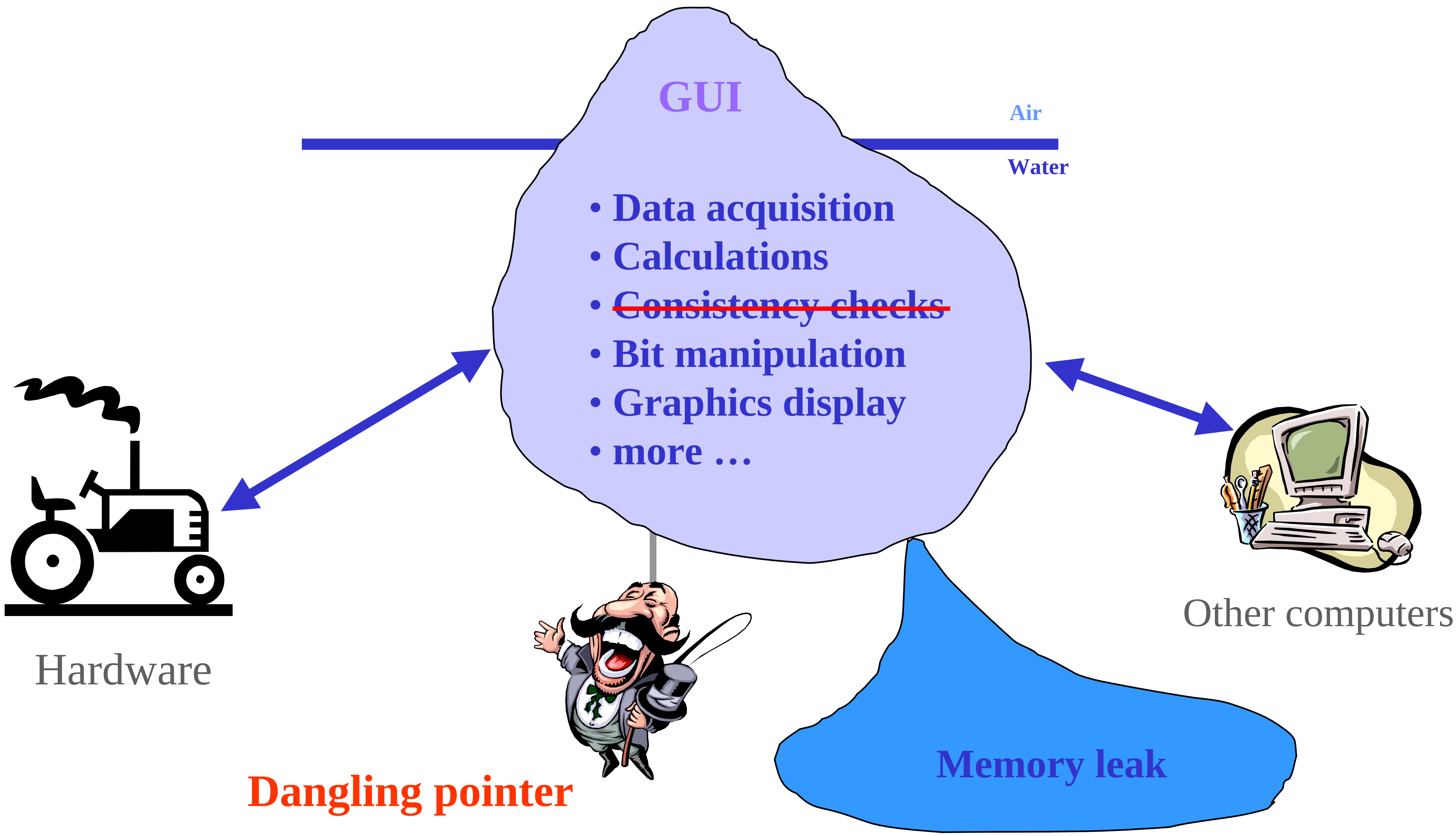
Cartoon view of BlackBox

Just what we need: a reasonable GUI supported by a strong engine



Cartoon view of C/C++

GUI supported by a strong engine that is ready to explode



BlackBox architecture

Client/server

Dynamic loading on-demand

Subsystems, modules, components

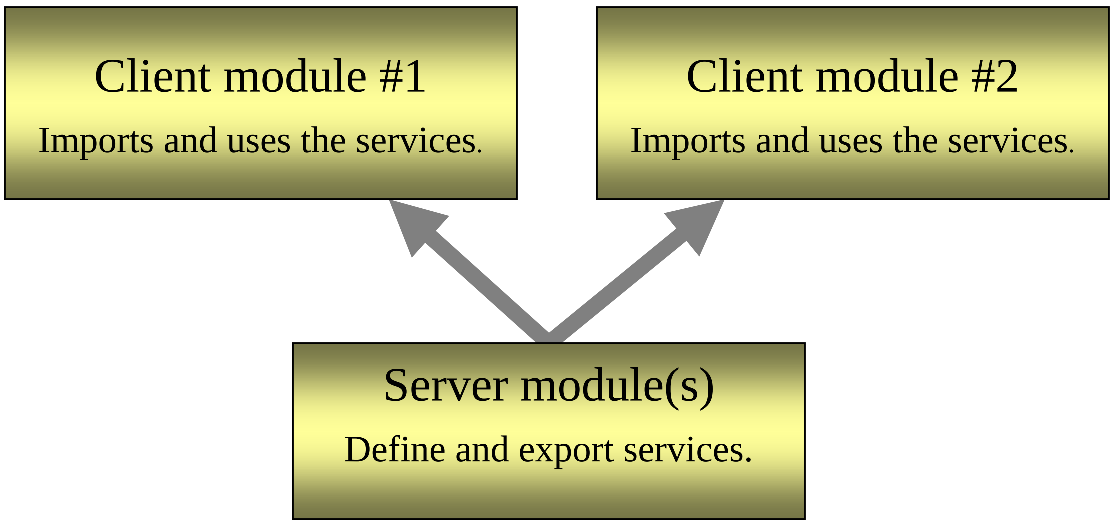
(direct support for hierarchical structure)

Subsystems, modules, components, objects

- Modules: units of compilation.
  - Also units of information hiding, and units being loaded to memory.
  - Physically, a module is usually a single file.
- Subsystems: units of development.
  - A subsystem is a collection of logically related modules.
  - A subsystem occupies its own directory tree.
- Components: units of deployment.
  - A component is a collection of modules or subsystems that you distribute.
  - A component is not a finished application. It is rather a part to be re-used.
- Objects: a programming concept.
  - An object is a data structure + related procedures (methods).
  - Objects do not take a center stage.

BlackBox has client/server architecture

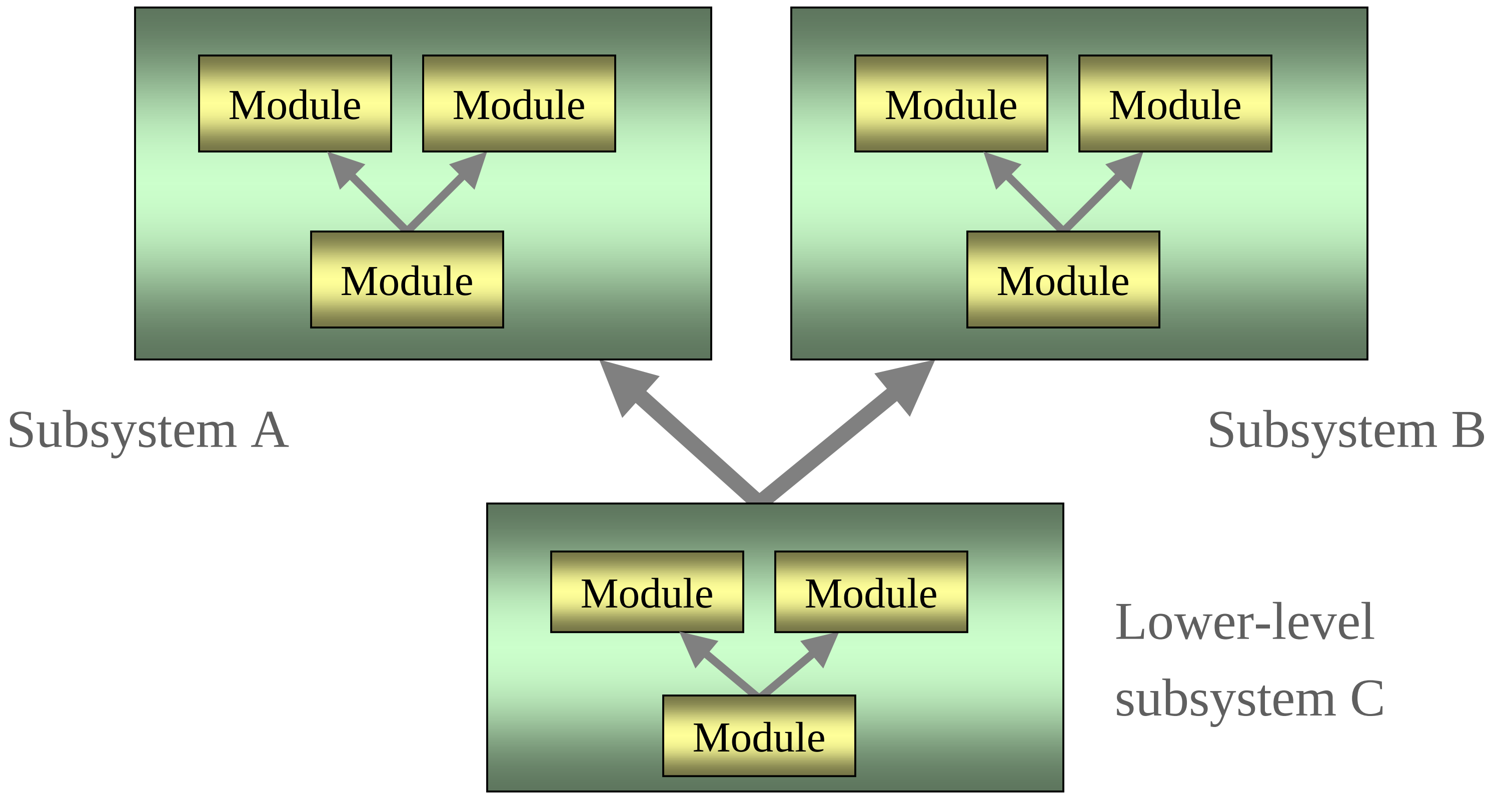
- **Servers:** Lower-level modules define and implement certain services.
- **Clients:** Higher-level modules make use of the services.
- Client/server relation is uni-directional (i.e., non-recursive).



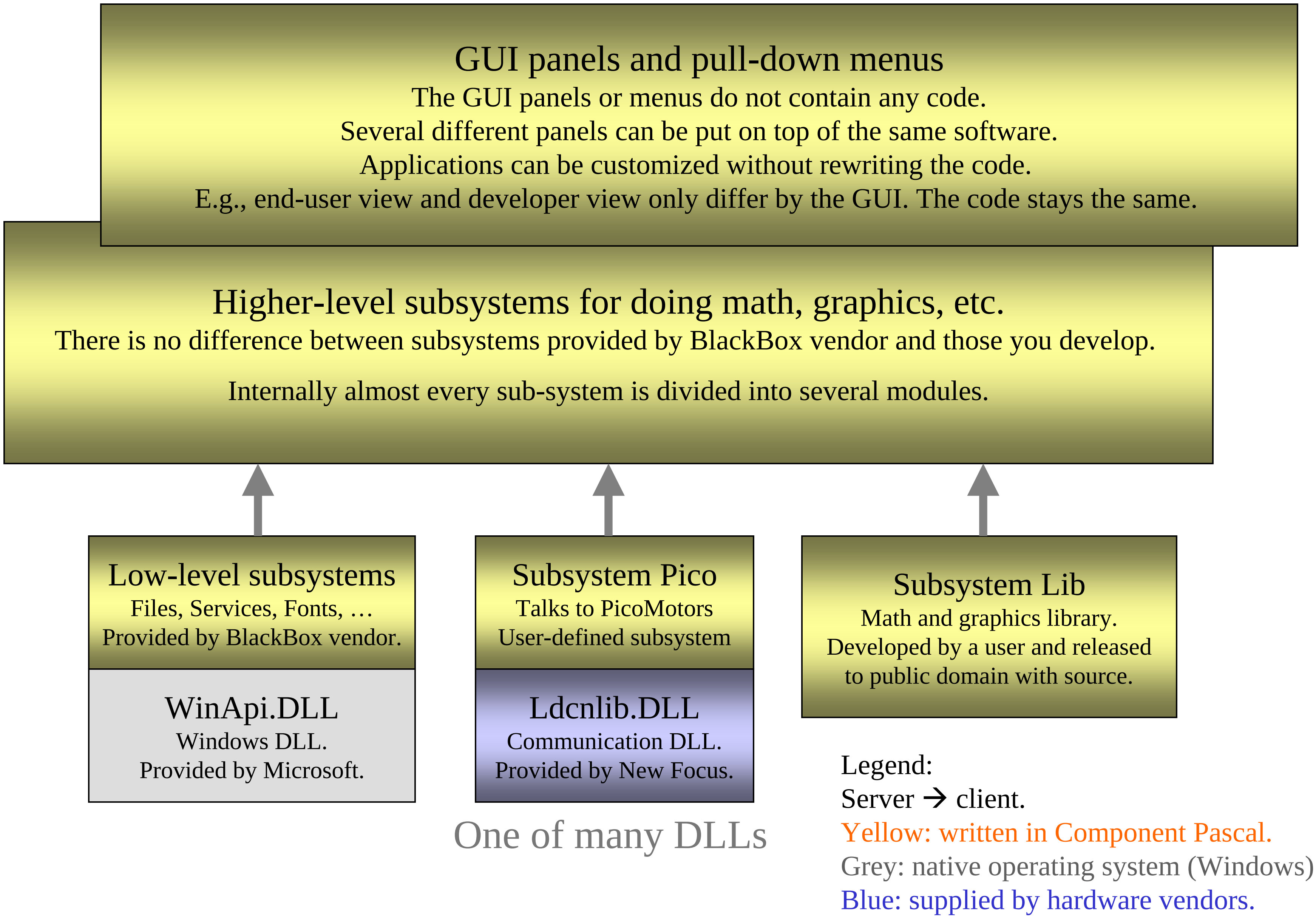
- Server defines and exports its services using a prescribed syntax.
- Client imports and makes use of the services.
- The compiler enforces consistency between the server and the client.
- The programmer cannot defeat the interfaces.

Modules are grouped into subsystems

- Related modules are grouped into subsystems.
- Subsystems live in their separate directory trees.
- Subsystems help better organize SW projects.
- Subsystems impose useful bookkeeping structure.



Subsystems help better organize complex projects



Modules are dynamically loaded on demand

All modules are equal under BlackBox (in particular, core BB modules).

LoadedModules

module name	bytes used	clients	compiled	loaded	Update
TextCmds	15618	0	04/29/2004 1:52:34 PM	12/10/2004 2:22:50 PM	
TextMappers	10790	8	04/29/2004 1:52:34 PM	12/10/2004 2:21:58 PM	
TextViews	26799	19	04/29/2004 1:52:33 PM	12/10/2004 2:21:58 PM	
TextSetters	29523	20	04/29/2004 1:52:33 PM	12/10/2004 2:21:58 PM	
TextRulers	28437	21	05/22/2001 2:10:28 PM	12/10/2004 2:21:58 PM	
TextModels	66557	45	05/22/2001 2:10:28 PM	12/10/2004 2:21:58 PM	
TgcCmd	10000	0	12/03/2004 4:51:26 PM	12/10/2004 2:22:02 PM	
TgcUnits	17832	1	12/08/2004 5:27:04 PM	12/10/2004 2:22:02 PM	
TgcGeometry	796	1	12/03/2004 5:03:54 PM	12/10/2004 2:22:02 PM	
TgcFilters	803	2	11/04/2004 2:05:59 PM	12/10/2004 2:22:02 PM	
TgcFarFields	7051	2	11/19/2004 3:10:38 PM	12/10/2004 2:22:02 PM	
TgcTools	14886	3	11/19/2004 4:03:11 PM	12/10/2004 2:22:02 PM	
LibSinhAxis	3076	4	02/11/2004 1:53:29 PM	12/10/2004 2:22:02 PM	
LibPolyPlotters	7811	4	02/11/2004 1:53:29 PM	12/10/2004 2:22:02 PM	
LibPlot3D	24863	4	02/11/2004 1:53:28 PM	12/10/2004 2:22:02 PM	
LibSolve	19875	5	02/11/2004 1:53:24 PM	12/10/2004 2:22:02 PM	
LibBessel	19520	1	02/11/2004 1:53:30 PM	12/10/2004 2:22:02 PM	
LibRealFn	9700	2	02/11/2004 1:53:25 PM	12/10/2004 2:22:02 PM	
LibEigenSys	22269	1	02/11/2004 1:53:25 PM	12/10/2004 2:22:02 PM	
LibCMatrices	15507	7	02/11/2004 1:53:24 PM	12/10/2004 2:22:02 PM	
LibAutoPlot	3571	2	02/11/2004 1:53:29 PM	12/10/2004 2:22:02 PM	

BlackBox Text subsystem

My own subsystem

Engineering subsystem (by R.Campbell)

Extensibility on the fly

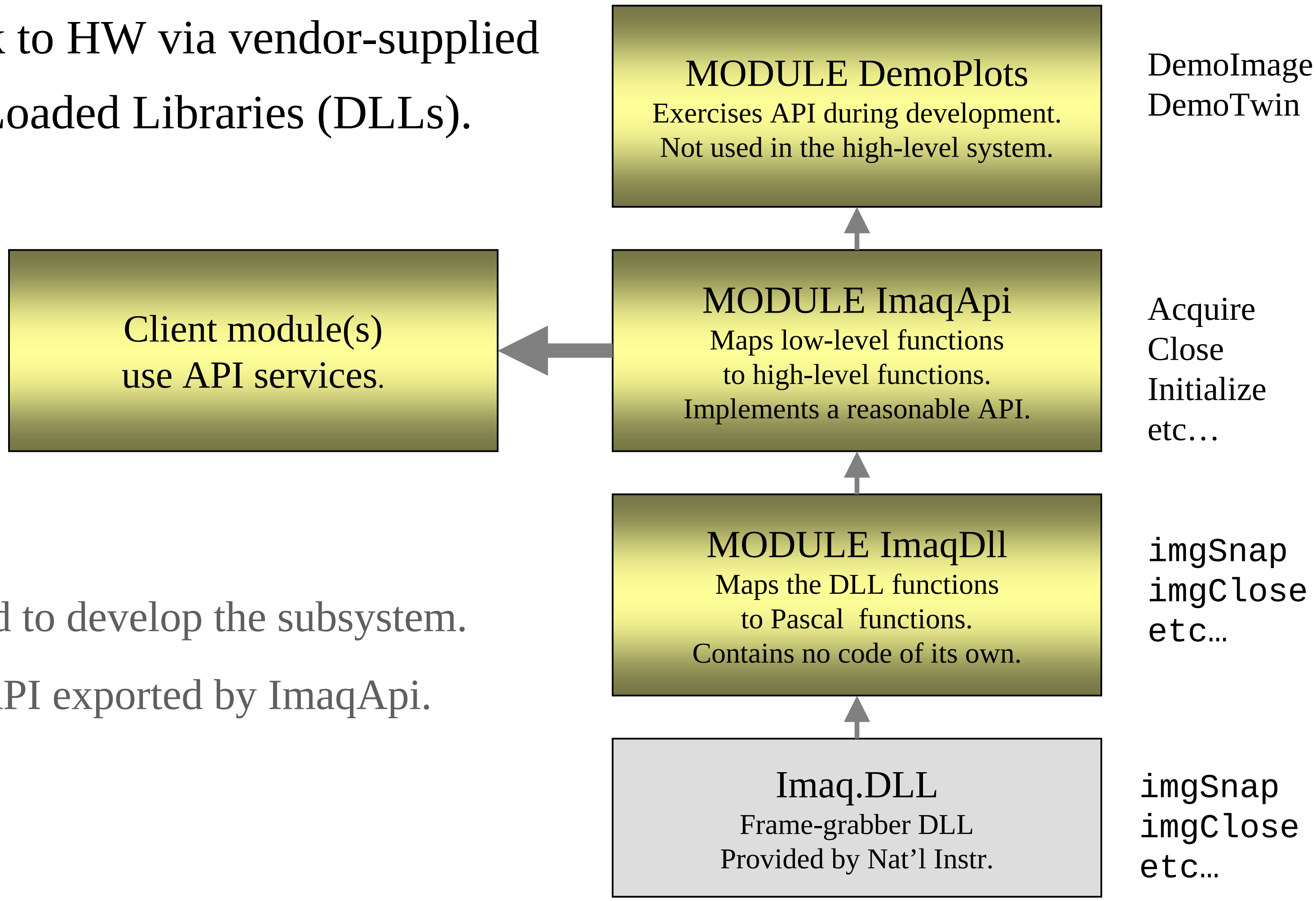
- All BlackBox components are dynamically loaded on demand.
  - Exception: the lowest level BB kernel and loader stay locked in memory.

- On-the-fly development cycle.
- Load – test – unload from memory.
- Redesign – modify source – recompile – load and test again.
- The cycle can be repeated many times.

- Compilation is instantaneous (fraction of a second for a fairly large module).
- Loading/unloading is instantaneous.
- All the interfaces are checked for consistency during compilation.
- The interface “fingerprints” are tested for consistency during loading.

Typical structure of a low-level subsystem that talks to HW

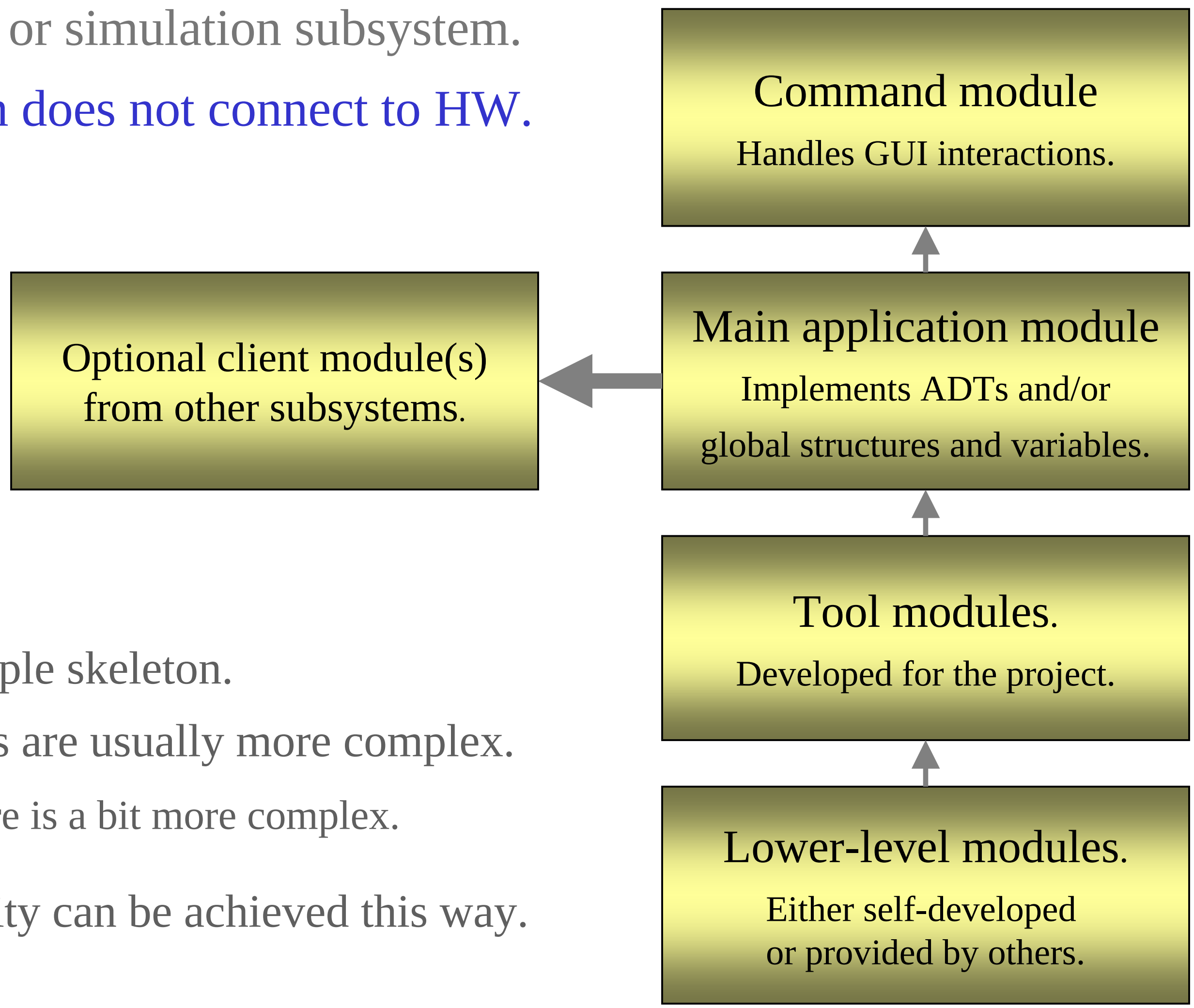
- This subsystem connects to HW.
- We always talk to HW via vendor-supplied Dynamically Loaded Libraries (DLLs).



- DemoPlot is used to develop the subsystem.
- Clients use the API exported by ImaqApi.

Typical structure of an application subsystem

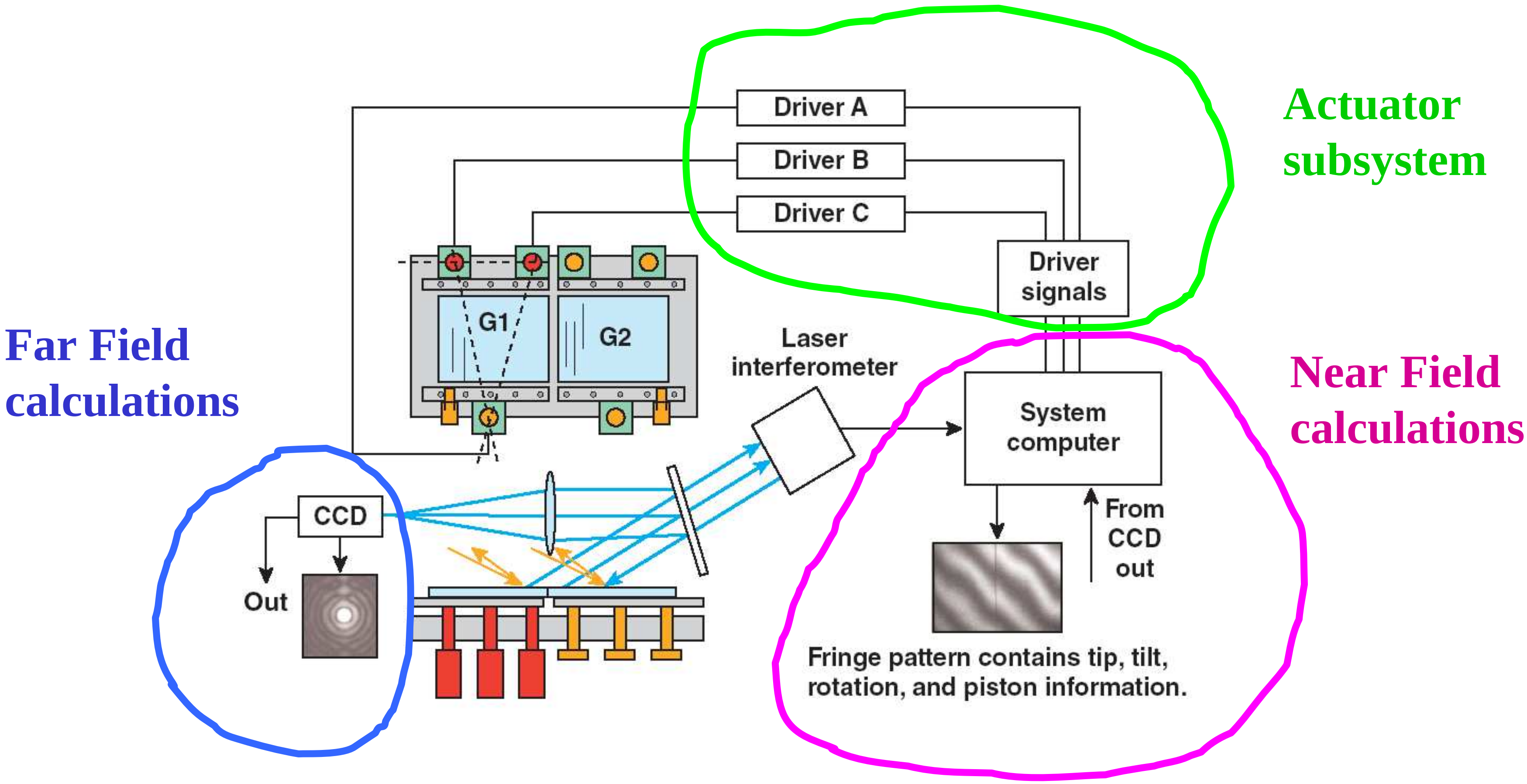
- A typical math or simulation subsystem.
- This subsystem does not connect to HW.



- This is a typical simple skeleton.
- Practical subsystems are usually more complex.
    - Our LLE software is a bit more complex.
- Amazing functionality can be achieved this way.

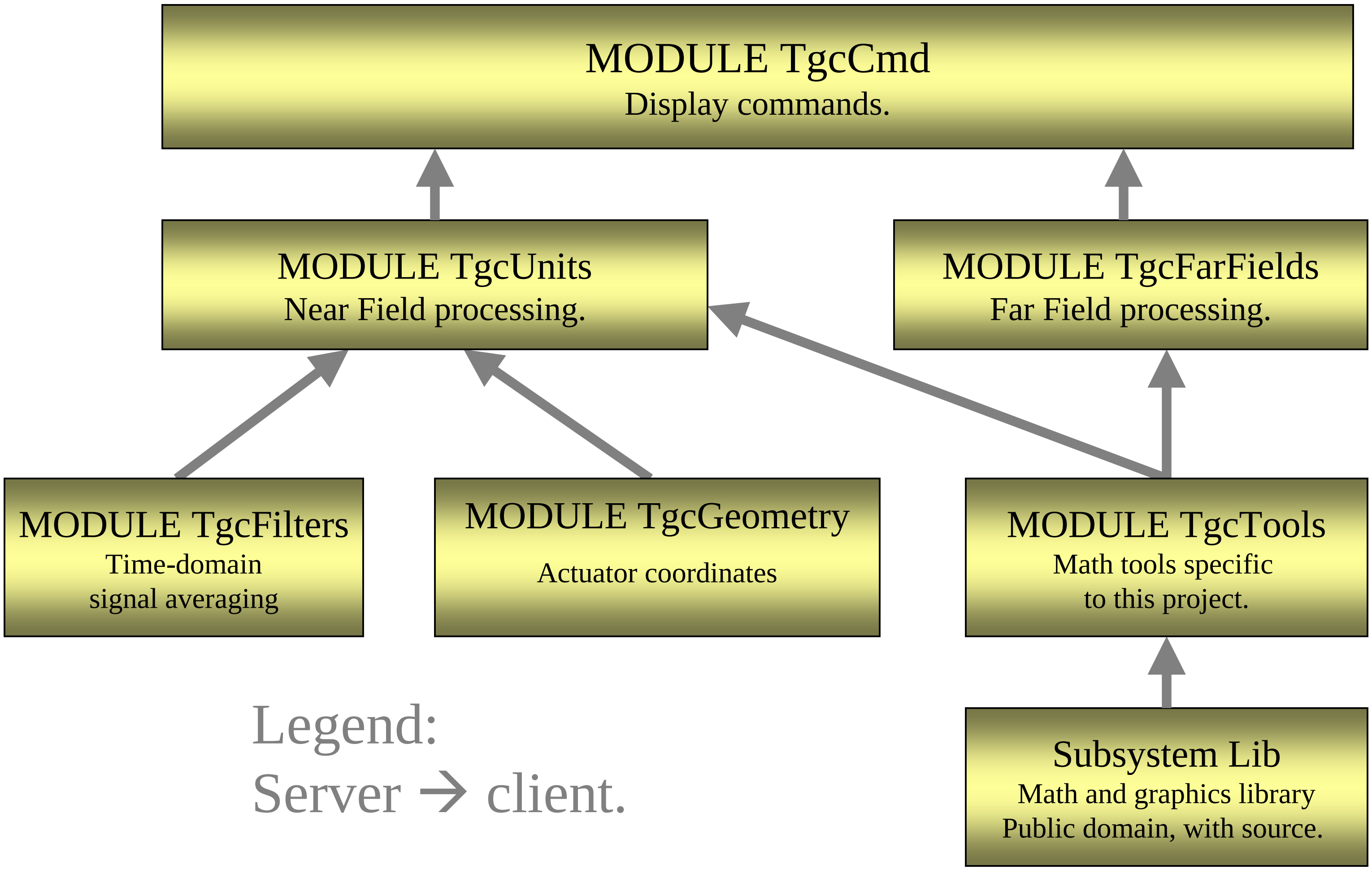
Software should follow the application logic

Next slide shows close correspondence between our TGA hardware and the architecture of our software. The correspondence is not exactly one-to-one, but it is close.



Application structure resembles the structure of HW

The diagram shows part of the structure of our LLE tiling software.



Legend:
Server → client.

Elements of the C.P. language

C.P. is a modern programming language.

It supports all major programming styles:

procedural, structured, object-oriented,

modular, and component-oriented.

C.P. is as safe as Java, but more efficient.

Safety features: garbage collection, safe typecasts, assertions, array index checking.

Complexity of syntax of programming languages

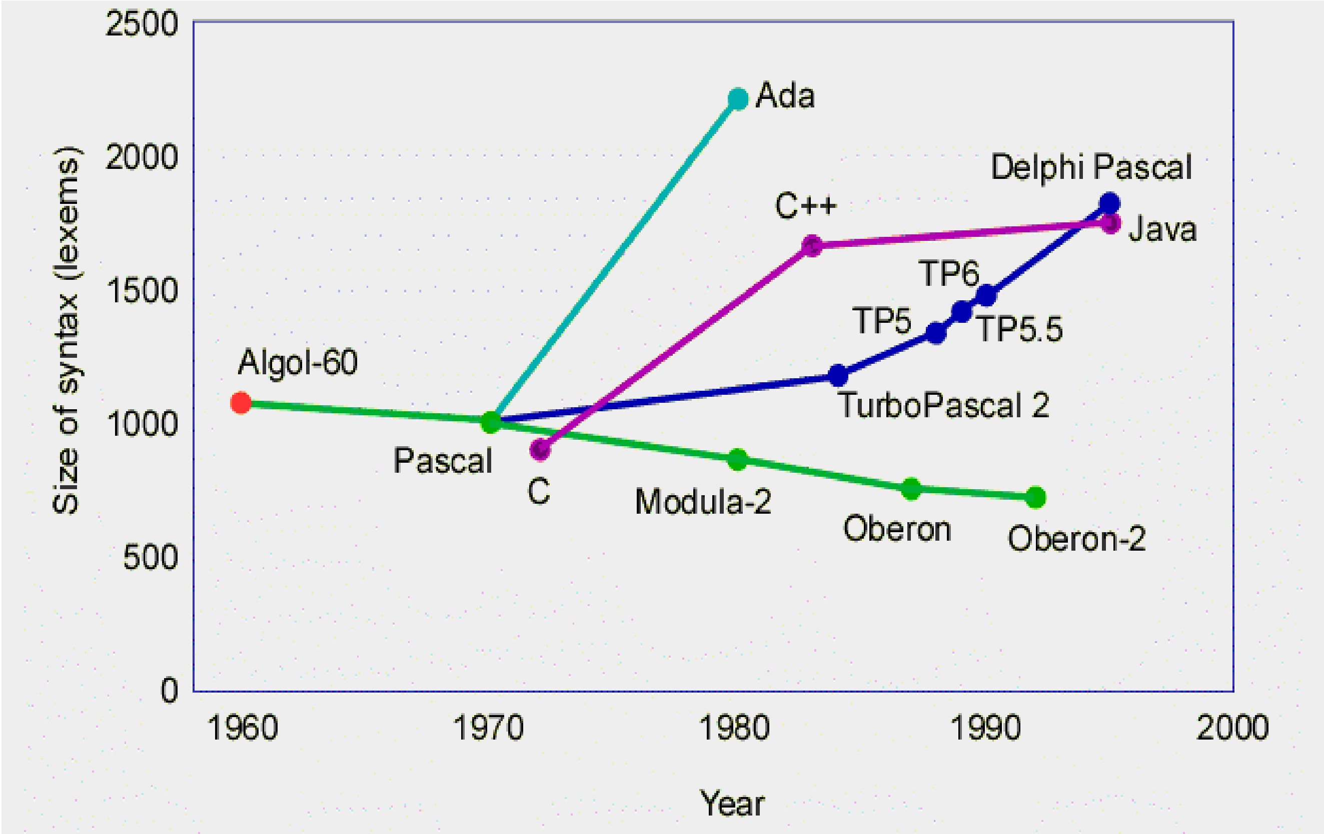


Figure 1. Complexity of syntax of programming languages

S.Z.Sverdlov (University of Vologda, Russia)

Source: N.Wirth lecture during the Oberon Day at CERN, www.cern.ch/Oberon.Day

Human-oriented structure is built into the C.P. language

The purpose of writing programs is communication.

Communication needs structure.

C.P. directly supports hierarchical structure.

Books	→	subsystems.
Chapters	→	modules.
Paragraphs	→	procedures and data structures.
Sentences	→	programming statements and variables.

C.P. uses English as its mother tongue

C.P. is intentionally based on human language to aid communication.

BEGIN … END, IF .. THEN .. ELSE, etc.

Syntax of operators is intended to catch single-letter typing mistakes.

E.g., mistyping "=" instead of ":=" will be caught by the compiler.

Many subtleties (practiced over 40+ years) help write good programs.

E.g., operators OR and & have graphical form that suggests & being of higher precedence than OR. This helps a human reader.

C.P. arithmetic statements strongly resemble Fortran

```
REPEAT
    INC (i);
    ff     :=  (i * ff + p + q) / (i * i - mu2);
    c      :=  c * d / i;
    p      :=  p / (i - mu);
    q      :=  q / (i + mu);
    del    :=  c * (ff + r * q);
    sum    :=  sum + del;
    del1   :=  c * p - i * del;
    sum1   :=  sum1 + del1
UNTIL  (ABS (del) < (1. + ABS (sum)) * eps)  OR
       (i = maxIt);
```

Modules encapsulate and organize

```
MODULE TstExample4;              ← module Example1 from the subsystem Tst

  IMPORT Math;                   ← lower-level server module Math is used

  VAR                            ← global declarations start here
    i * : INTEGER;               ← global read/write variable
    j - : INTEGER;               ← global read-only  variable
    r   : REAL;                  ← global hidden     variable (i.e., private)


  PROCEDURE Tweak * ;            ← procedure Tweak is exported
    BEGIN
      r := Math.Sin (i * 3.14);  ← imported procedure Sin is used here
    END Tweak;


  PROCEDURE Sub  ;               ← hidden procedure, note absence of * mark
    BEGIN
      j := i - 1;
    END Sub;

END TstExample4.                 ← compilation stops at the fullstop "."
```

Variables and assignments

Arithmetic variables can be assigned values, <u>if it is safe to do so</u>.

No loss of information → assignment operation is allowed.
Loss of information → explicit conversion must be made.

```
REAL    := INTEGER is allowed.
INTEGER := REAL is allowed with explicit conversion.
```

Similar rules apply in case of data structures and pointers.

"Assignment compatibility" is enforced in order to avoid losing information.

Descriptive error makers are used by BlackBox, see next slide.

Variables and assignments

Example5

```
MODULE TstExample5;     (*Data types and assignment rules*)
VAR
  b : BYTE;             (*signed  8 bit integer*)
  k : SHORTINT;         (*signed 16 bit integer*)
  i : INTEGER;          (*signed 32 bit integer*)
  j : LONGINT;          (*signed 64 bit integer*)
  p : SHORTREAL;        (* 32 bit floating point*)
  r : REAL;             (* 64 bit floating point*)
  s : SET;              (* 32-bit bit pattern*)
  flag : BOOLEAN;       (* bool variable, TRUE | FALSE*)
BEGIN
  s := BITS (i);        (* bit pattern <== integer, OK*)
  i := ORD  (s);        (* reverse of the above, OK*)

  r := j;               (* real    <== integer, OK*)
  j := ENTIER (r) ;     (* integer <== real, OK explicit trunc*)
  j := r  ;             (* integer <== real, not OK*)

  k := flag  ;          (* integer <== boolean, not OK*)

  i := k;                         (* 32-bit <== 16-bit, OK*)
  k := SHORT(i);         (* 16-bit <== truncation of 32-bit, OK*)
  k := i incompatible assignment; (* 16-bit <== 32-bit, not OK*)

END TstExample5.
```

Arrays and data structures

Multidimensional arrays of arbitrary elements.

Every array carries a hidden "tag" with range information. The run-time system uses the "tag" information to perform index range checking.

```
arr1 = ARRAY 100 OF INTEGER;
arr2 = ARRAY 100, 100 OF REAL;
```

Data structures composed of arbitrary elements.

Every structure carries a hidden "tag" with type information. The run-time system uses the "tag" information to enforce consistency of assignments.

```
struct1 = RECORD i: INTEGER; r: REAL END;
struct2 = RECORD a1: arr1;  a2: arr2 END;
```

Both the arrays and data structures are "tagged" to enforce run-time consistency.

Pointers and memory management

Pointers and pointer arithmetic are the leading causes of catastrophic errors in unsafe programming languages (C and C++). Consequently, pointers are implemented with great care in C.P. Such restricted pointers are named "references" in other languages.

- Pointer arithmetic is not supported.
- Pointers to scalar variables are not supported.
- Only pointers to arrays or to data structures are supported*.
- Dynamically created variables cannot be de-allocated "by hand".
- The run-time "garbage collector" maintains the dynamic memory pool.

```
aPtr  = POINTER TO ARRAY 100 OF INTEGER;
sPtr  = POINTER TO RECORD a1, a2: aPtr  END;
NEW (sPtr);  NEW (sPtr.a1);  NEW (sPtr.a2);
```

*Arrays and structures are "tagged" in order to guard pointer safety.

Memory management and "garbage collection"

Dynamically created arrays and data structures cannot be de-allocated "by hand".

The run-time "garbage collector" automatically maintains the dynamic memory pool.

Example6

```
MODULE TstExample6;  (*Memory management and garbage collection*)
VAR
   arr : POINTER TO ARRAY OF REAL;
   i   : INTEGER;

BEGIN

  NEW (arr, 100);        (* create the array with 100 elements *)

  FOR i := 0 TO LEN (arr) - 1 DO                (* loop 0 .. 99 *)
    arr [i] := 10.0 * i
  END;

  arr := NIL;                  (* discard the pointer to array *)

  (* garbage collector will reclaim 800 bytes of storage *)

END TstExample6.
```

Modules

Modules are the building blocks.

Modules fit to each other (like LEGO blocks).

Modules encapsulate their content:

variables, procedures, and data structures.

BlackBox is composed of modules

- BlackBox is a collection of subsystems.
- Subsystems are built of modules.
- Modules fit to each other (like LEGO blocks).
- Modules encapsulate and organize their content: variables, procedures, and data structures.

A simple module

```
MODULE TstExample1;                    ← module Example1 from the subsystem Tst

  IMPORT StdLog;                       ← lower-level server module StdLog is used

  VAR i: INTEGER;                      ← global hidden variable


  PROCEDURE Add * ;                    ← exported procedure
    BEGIN
      StdLog.String("Old i = ");
      StdLog.Int(i); StdLog.Ln;
      i := i + 1;
      StdLog.String("New i = ");
      StdLog.Int(i); StdLog.Ln;
    END Add;


  PROCEDURE Sub  ;                     ← hidden procedure, note absence of * mark
    BEGIN
      i := i - 1;
    END Sub;

END TstExample1.                       ← compilation stops at the fullstop "."
```

The interface of the simple module

```
DEFINITION TstExample1;

      PROCEDURE Add;

END TstExample1.
```

- The “definition” was created automatically by BlackBox.
- Non-exported entities are hidden, i.e., protected from access.
- Both “i” and “Sub” are hidden and therefore not accessible.
- “Information hiding” is the most important role of modules.

A modified module

```
MODULE TstExample1;                ← module Example1 from the subsystem Tst

  IMPORT StdLog;                   ← lower-level server module StdLog is used

  VAR i * : INTEGER;               ← global variable “i” is now exported


  PROCEDURE Add * ;                ← exported procedure
    BEGIN
      StdLog.String("Old i = ");
      StdLog.Int(i); StdLog.Ln;
      i := i + 1;
      StdLog.String("New i = ");
      StdLog.Int(i); StdLog.Ln;
    END Add;


  PROCEDURE Sub * ;                ← procedure Sub is now exported
    BEGIN
      i := i - 1;
    END Sub;

END TstExample1.                   ← compilation stops at the fullstop “.”
```

The interface of the modified module

```
DEFINITION TstExample1;

       VAR
              i: INTEGER;        ← global variable "i" is now exported

       PROCEDURE Add;
       PROCEDURE Sub;            ← procedure Sub is now exported

END TstExample1.
```

- During compilation the following message was shown in the Log console.

```
compiling "TstExample1"
  Sub is new in symbol file
  i is new in symbol file
```

Interface defines the module's services

- Every module has an "interface" that tells what services the module is offering to other modules.

- The interface can be modified by either exporting what was hidden, or hiding what was exported. After recompilation, the new interface takes effect.

- After the interface was changed, all clients have to be recompiled.

- The "contract" between the server and the client is checked by the compiler. The client cannot be compiled unless the contract is fulfilled.

Where does the program execution start?

- Modules can have multiple entry points.
- Every entry point is as good as any other.
- Where is the function **`main`** in this system? Nowhere.
- Execution can start at any entry point.
- SW can be executed piece-wise.
- Great for step-wise development.
- Top-level modules are "servers to the user".

  Top-level modules do not have other modules as explicit clients. The user can call the entry points of the top-level modules by clicking. Therefore, top-level modules serve the user.

Example of the execution trail

BlackBox

File Edit Attributes Info Dev Tools Controls Text SQL Obx Tut Ctls GrPlotter GrDAQDemo Infodesk Multi Pac Tgc Wands Window Help

Log

```
compiling "TstExample1"
  new symbol file   152   4
compiling "TstExample1"
  Sub is new in symbol file
  i is new in symbol file   160   4

Old i =  0
New i =  1          Add

Old i =  1          Add
New i =  2

Old i =  2          Add
New i =  3


Old i =  3          Sub
New i =  2

Old i =  2          Sub
New i =  1
```

ToDo.txt

```
MODULE TstExample1;
  IMPORT StdLog;
  VAR i * : INTEGER;
  PROCEDURE Add * ;
    BEGIN
      StdLog.String("Old i = "); StdLog.Int(i); StdLog.Ln;
      i := i + 1;
      StdLog.String("New i = "); StdLog.Int(i); StdLog.Ln;
  END Add;

  PROCEDURE Sub * ;
    BEGIN
      StdLog.String("Old i = "); StdLog.Int(i); StdLog.Ln;
      i := i - 1;
      StdLog.String("New i = "); StdLog.Int(i); StdLog.Ln;
  END Sub;

END TstExample1.

! TstExample1.Add
! TstExample1.Sub

Points:
 -- this module is a client of StdLog (imports it and uses its services)
 -- syntax to specify client/server relation is simple and unambiguous
 -- this module is a server as well (it does some things, presumably useful)
 -- some things are hiden, some are exported
 -- hidden can be made visible
 -- visible can be made hidden
 -- before the changes take effect, old version has to be unloaded
 -- devel GUI can be embedded in the same file as the module
 -- GUI can be made automatically
```

Objects and data structures

Data structures help organize heterogeneous data.

Objects = data structures + “methods”.

Objects do not take the center stage.

Objects are useful, but they are not holy cows.

Choose an appropriate programming style

A few years ago, there was an "objectomania" among programmers.

To this day we hear that "everything should be written as classes in C++".

Objects are useful, but should not become a gospel.

Component Pascal supports all major styles of modern programming.

Computational libraries are best written in classic procedural style.

Any Numerical Recipe can be immediately translated into C.P.

Complex applications need data structures, but not necessarily objects.

If a given hardware item has complex behaviors, then we may add "methods".

An example: a camera can acquire data, hence Camera.Acquire.

A camera thus becomes an "object".

Choose a programming style that suits your needs and your skill level.

A data structure

```
TYPE                                   type of export: R/W, read-only, or none
 Camera * = RECORD
             chan       *  : INTEGER;  (*frame grabber channel*)
             err        -  : INTEGER;  (*most recent error code*)
             bufsize    -  : INTEGER;  (*size of the buffer for image data*)
             bufAdr     -  : INTEGER;  (*address of the buffer*)
             update     *  : BOOLEAN;  (*do you want to update the screen?*)
             iid, sid      : INTEGER;  (*interface and session ID, both hidden*)
           END;
```

Heterogeneous data items are grouped into one convenient data structure that can be manipulated as a whole. The name of such beast in C would be `struct`.

The markers * and - make the items exported (i.e., accessible for the clients) as read/write, read-only, or not at all (if there is no marker).

N.B. This particular data structure is declared statically, i.e., it does not have to be explicitly allocated by calling NEW.

This is named a "stack variable" in computer jargon.

An object = data structure + methods

```
TYPE                                      dynamically allocated
  Unit * = POINTER TO RECORD

    cam          -  :  ImaqApi.Camera; (*previously defined data structure*)
    showFringes  *  :  BOOLEAN;        (*a data item*)

    (u: Unit) AcquireImage * ;      procedures "bound to" this particular type
    (u: Unit) InitDaq      * ;
    (u: Unit) InitDrift    * ;
END;
```

A data structure that has procedures bound to it is termed object. The name of such beast in C++ would be `class`.

The "data items" would be termed `members` in C++. Note that an entire data structure Camera can also be a data item!

N.B. This particular data structure is declared dynamically, i.e., it has to be explicitly allocated by calling NEW.

This is named a "heap variable" in computer jargon.

Gr waveform graphics*

Histogram and waveform display package.

Fairly easy to use.

Designed for data acquisition (DAQ).

Used several times for DAQ display.

DDC-1 and DDC-8 waveform digitizers, CAMAC, and student projects.

*Developed by W.S.

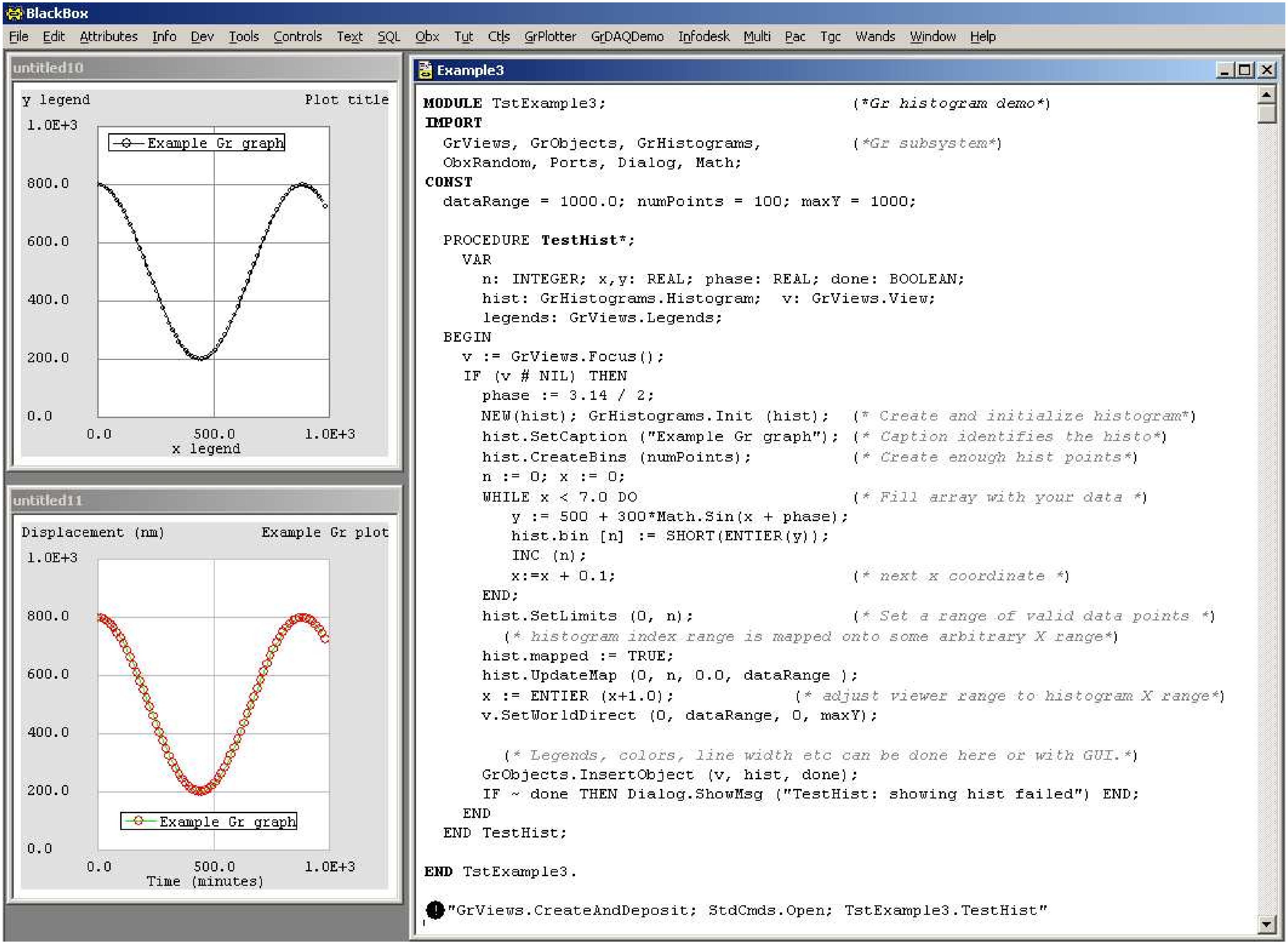
When to use Gr (and when not)

When to use Gr:

- Integer-valued data, e.g., acquired from a digitizer (32-bit integers).
- One-dimensional data. E.g. temperature recordings, etc.
- The data is not an image (which would require 2D display).
- Real-valued math is not planned.
- The display has to be “live” and interactive.
- Convenient zooming and scaling is needed.

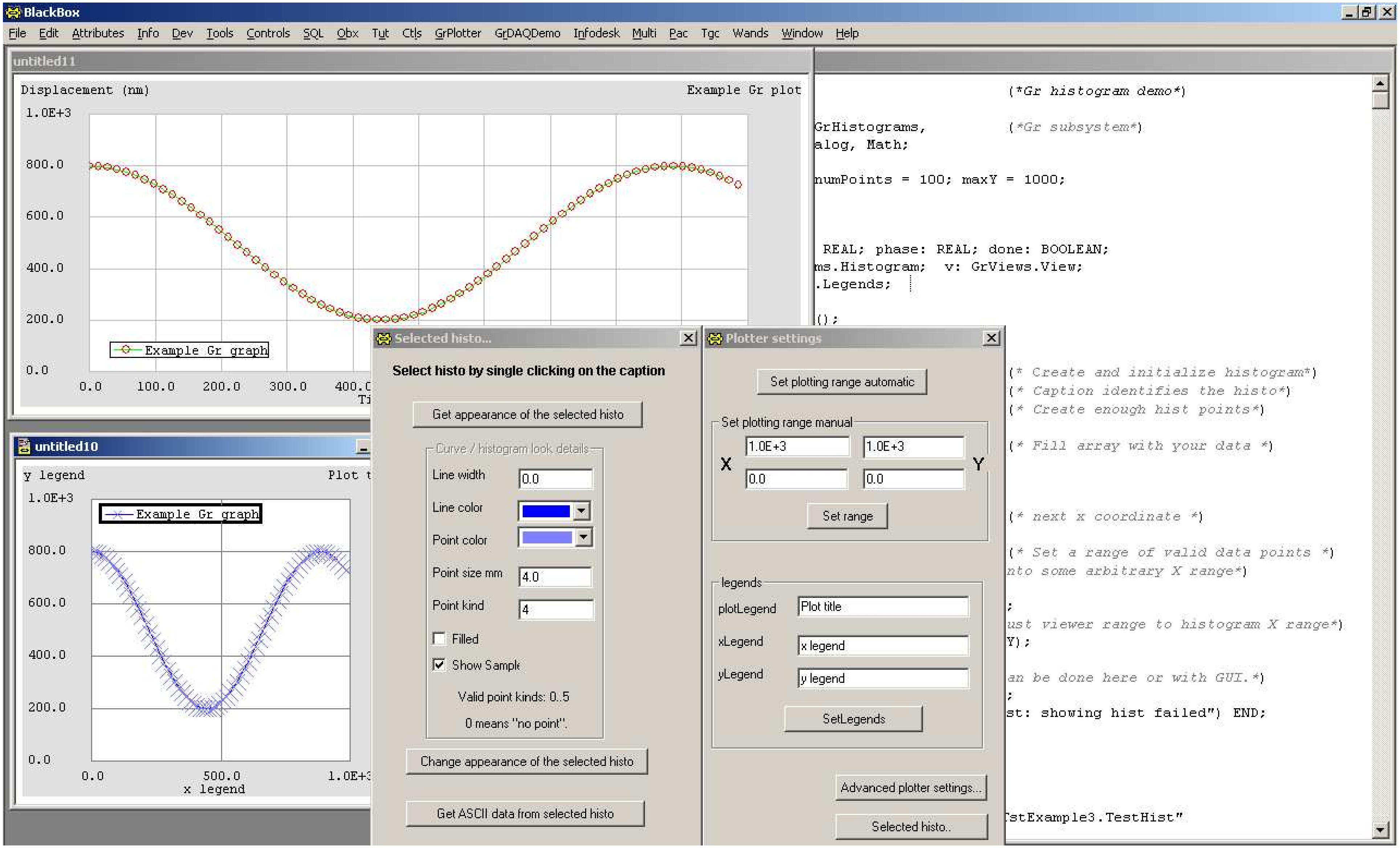
When to use Lib rather than Gr:

- Real-valued data (64-bit floating point numbers).
- Two-dimensional display (e.g., camera images).
- Lots of math, matrix algebra, or vector operations on data.

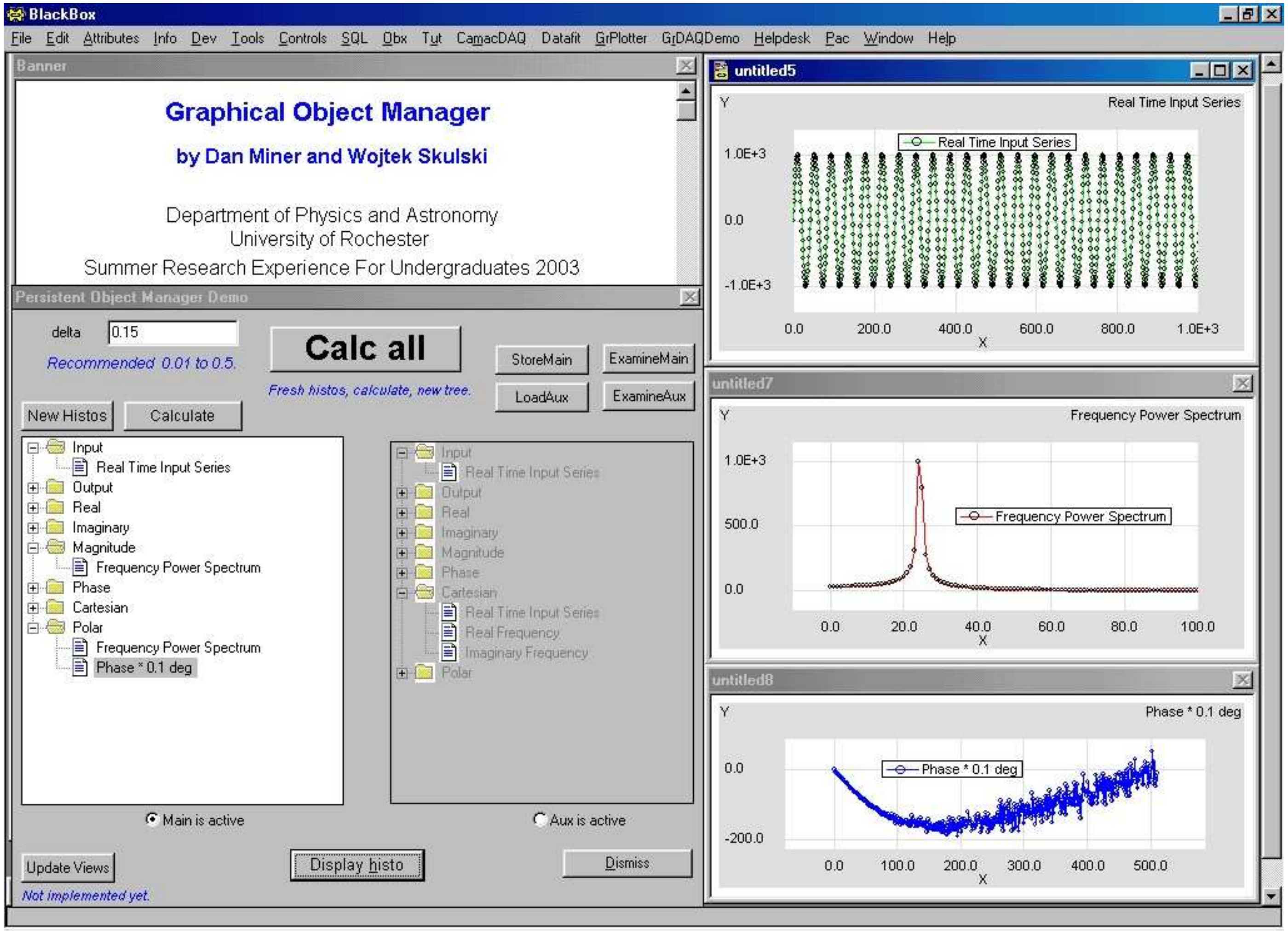

BlackBox
File Edit Attributes Info Dev Tools Controls Text SQL Obx Tut Ctls GrPlotter GrDAQDemo Infodesk Multi Pac Tgc Wands Window Help
untitled10
y legend
Plot title
1.0E+3
800.0
600.0
400.0
200.0
0.0
Example Gr graph
0.0
500.0
1.0E+3
x legend
untitled11
Displacement (nm)
Example Gr plot
1.0E+3
800.0
600.0
400.0
200.0
0.0
Example Gr graph
0.0
500.0
1.0E+3
Time (minutes)
Example3
MODULE TstExample3; (*Gr histogram demo*)
IMPORT
GrViews, GrObjects, GrHistograms, (*Gr subsystem*)
ObxRandom, Ports, Dialog, Math;
CONST
dataRange = 1000.0; numPoints = 100; maxY = 1000;
PROCEDURE TestHist*;
VAR
n: INTEGER; x,y: REAL; phase: REAL; done: BOOLEAN;
hist: GrHistograms.Histogram; v: GrViews.View;
legends: GrViews.Legends;
BEGIN
v := GrViews.Focus();
IF (v # NIL) THEN
phase := 3.14 / 2;
NEW(hist); GrHistograms.Init (hist); (* Create and initialize histogram*)
hist.SetCaption ("Example Gr graph"); (* Caption identifies the histo*)
hist.CreateBins (numPoints); (* Create enough hist points*)
n := 0; x := 0;
WHILE x < 7.0 DO (* Fill array with your data *)
y := 500 + 300*Math.Sin(x + phase);
hist.bin [n] := SHORT(ENTIER(y));
INC (n);
x:=x + 0.1; (* next x coordinate *)
END;
hist.SetLimits (0, n); (* Set a range of valid data points *)
(* histogram index range is mapped onto some arbitrary X range*)
hist.mapped := TRUE;
hist.UpdateMap (0, n, 0.0, dataRange);
x := ENTIER (x+1.0); (* adjust viewer range to histogram X range*)
v.SetWorldDirect (0, dataRange, 0, maxY);
(* Legends, colors, line width etc can be done here or with GUI.*)
GrObjects.InsertObject (v, hist, done);
IF ~ done THEN Dialog.ShowMsg ("TestHist: showing hist failed") END;
END
END TestHist;
END TstExample3.
"GrViews.CreateAndDeposit; StdCmds.Open; TstExample3.TestHist"

Gr has a reasonably good interactive GUI and is very easy to use.

A reasonably looking Gr plot can be obtained in less than a minute.



Graphical object management of Gr histograms



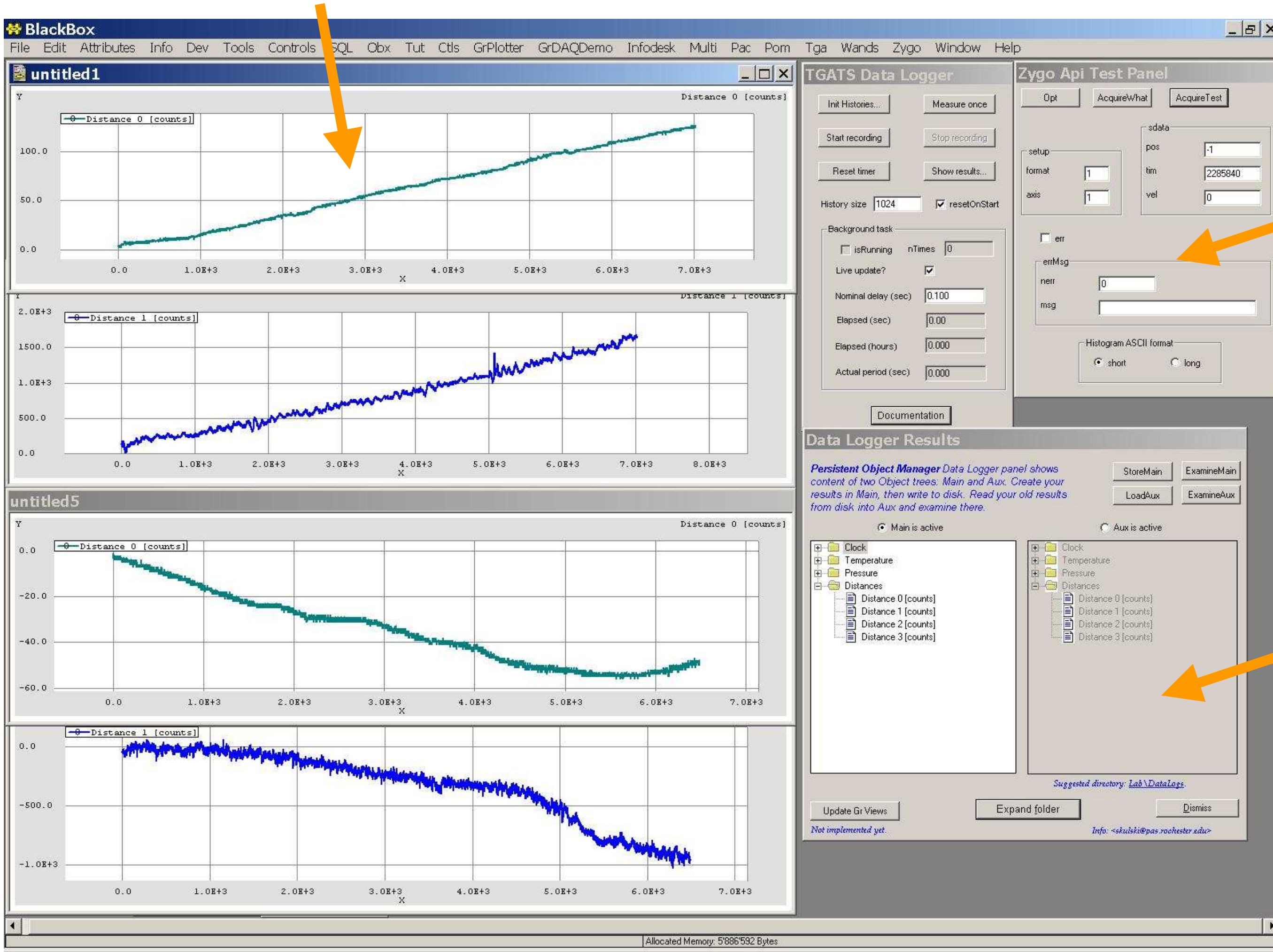
Gr/POM helps organize histograms into a tree structure.

Developed by an undergraduate student (with very little help)!

Data Logger shows interferometer data histories

Displacement-measuring interferometer

history recordings



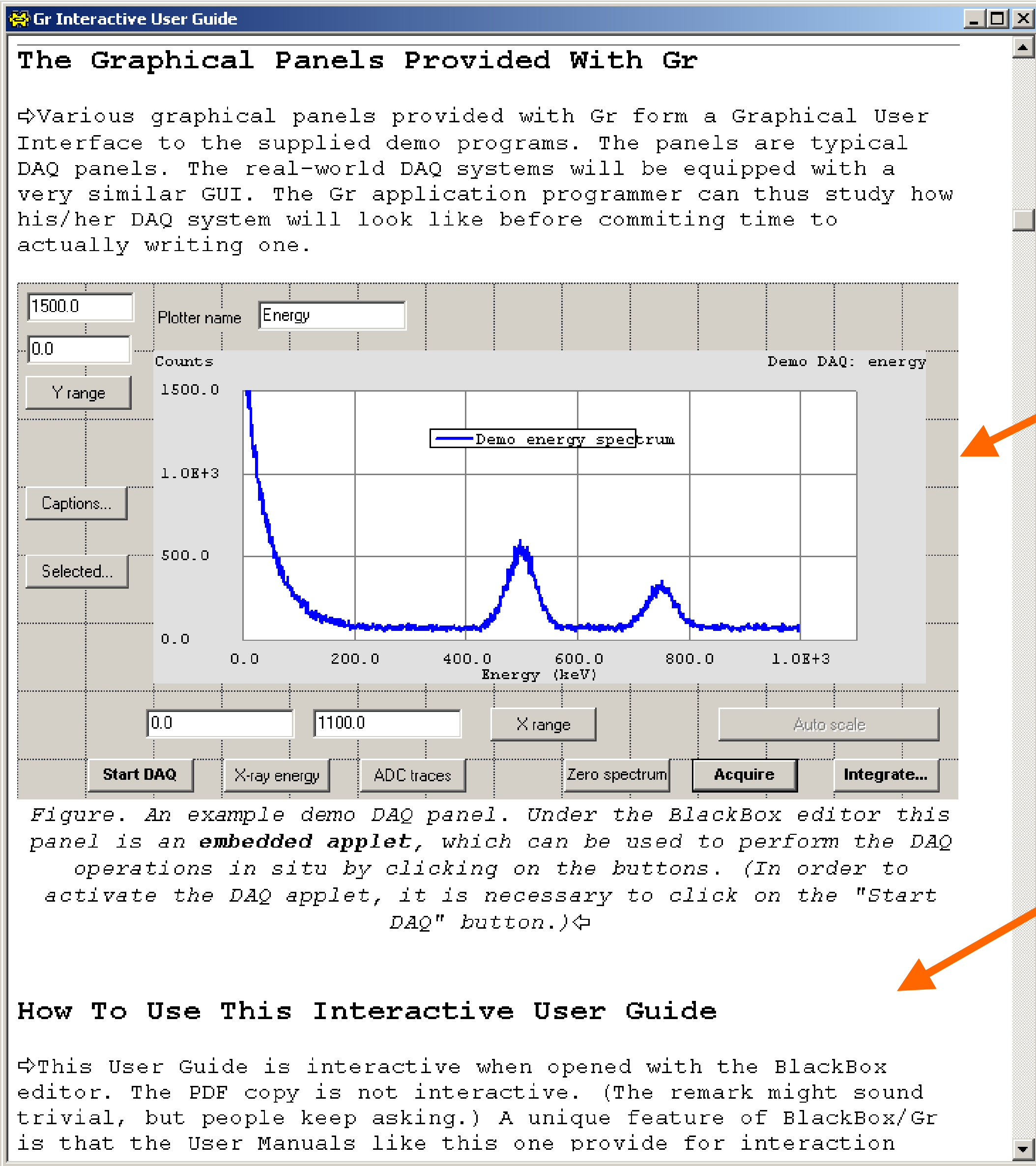
Data acquisition

panels

Data management

panel

Gr User Guide



Gr Interactive User Guide

The Graphical Panels Provided With Gr

Various graphical panels provided with Gr form a Graphical User Interface to the supplied demo programs. The panels are typical DAQ panels. The real-world DAQ systems will be equipped with a very similar GUI. The Gr application programmer can thus study how his/her DAQ system will look like before commiting time to actually writing one.

Figure. An example demo DAQ panel. Under the BlackBox editor this panel is an ***embedded applet****, which can be used to perform the DAQ operations in situ by clicking on the buttons. (In order to activate the DAQ applet, it is necessary to click on the "Start DAQ" button.)*

How To Use This Interactive User Guide

This User Guide is interactive when opened with the BlackBox editor. The PDF copy is not interactive. (The remark might sound trivial, but people keep asking.) A unique feature of BlackBox/Gr is that the User Manuals like this one provide for interaction

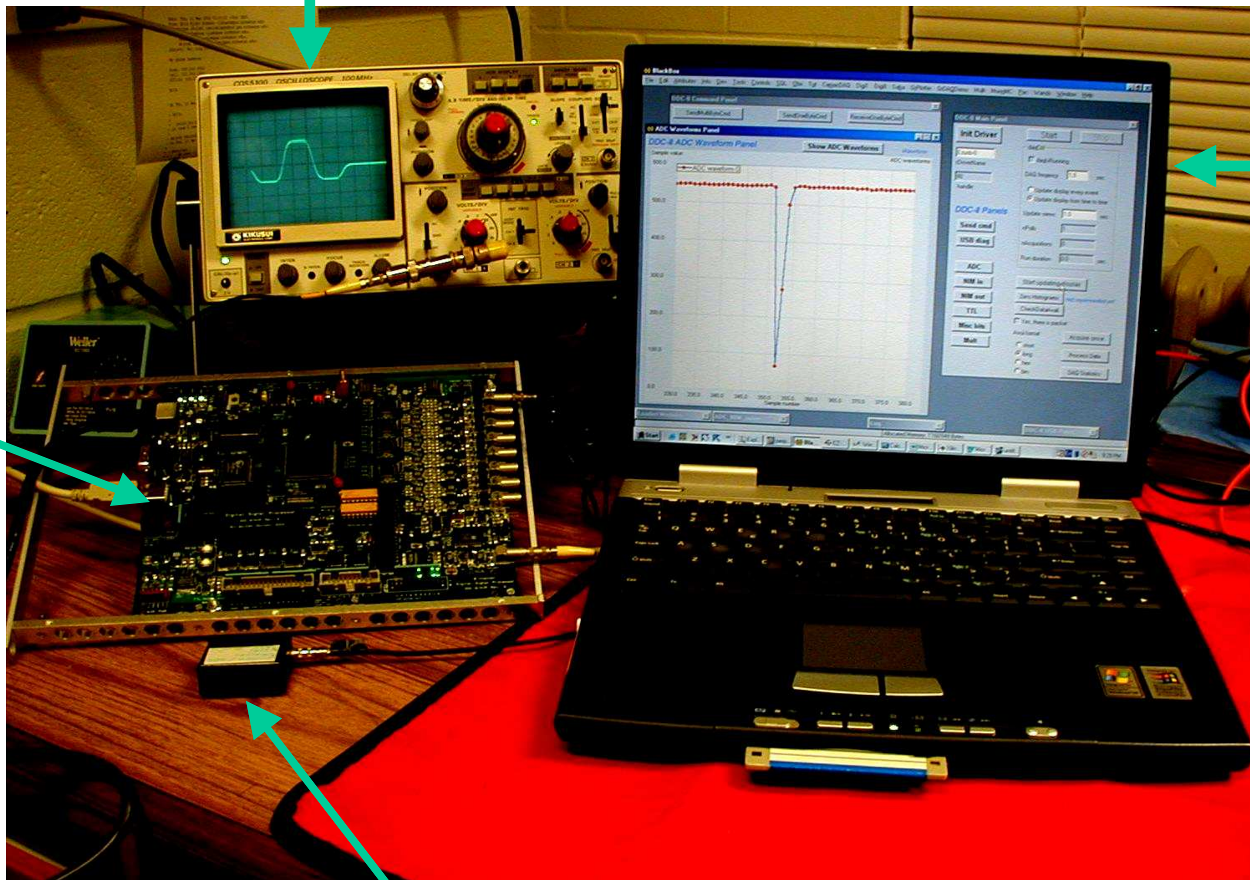
Example data acquisition panel.

Detailed instructions how to use the Gr subsystem.

Gr was designed for data acquisition. Several example panels and a tutorial are provided with the Gr package.

DDC-x development system using BlackBox and Gr

Analog signal reconstruction: digital FIR filter output



DDC-8

Control & waveform display: BlackBox and Gr

NIM pulser

Scientific and engineering library Lib*

1D and 2D display package.

Matrix and vector calculus, digital filters, and special functions.

Designed for data analysis, theoretical calculations, and modeling.

Extensively used for mission-critical applications.

Airborne antenna systems (BAE Systems), adaptive optics (UofR Laser Lab).

*Developed by Robert Campbell, BAE Systems.

When to use *Lib* libraries

When to use:

• Real-valued data (64-bit floating point numbers).

• Two-dimensional display (e.g., camera images).

• Lots of math, matrix algebra, or vector operations on data.

The list of subsystems:

Lib - Engineering & Scientific library, including 1D, 2D, and 3D plotting.

Algebra - Computational Algebra library.

Multi - Arbitrary ultra high precision arithmetic.

Filter - Digital and other filter design and analysis.

Nav - Navigation & Geodesic Coordinate transform modules and tools.

Wands - A collection of general purpose tools (Wand: a tool used by a magician).

Cpc and Ctls- several general-purpose tools and utilities.

Lib: math and plotting facilities

Vectors & Matrices, including 0-length cases.

Polynomial forms of vectors, matrix determinant, inverse, singular-value decomposition, etc…

Complex numbers and complex functions.

Special Functions & curve fitting.

Random numbers: several r.n. generators.

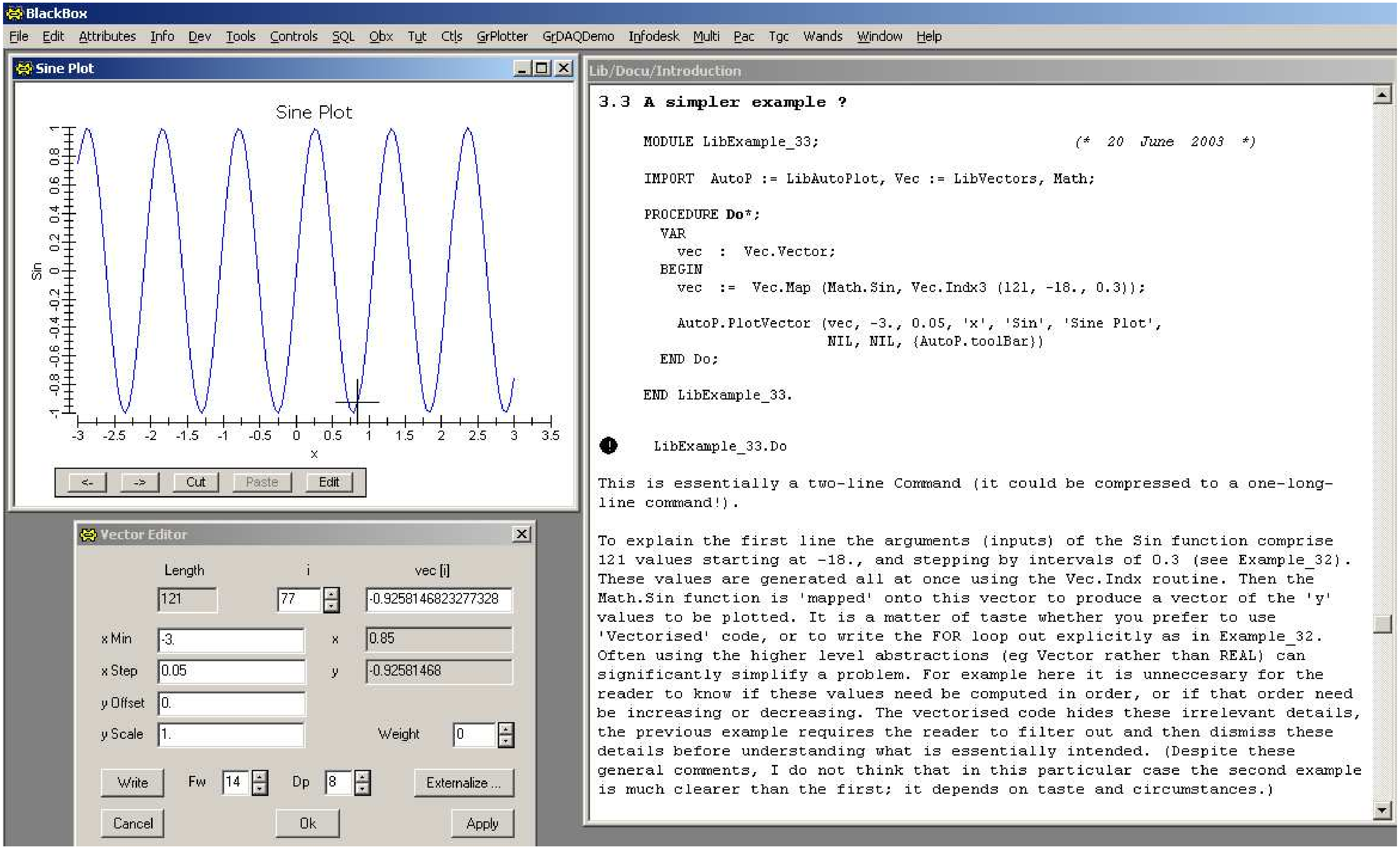
Numerical methods.

Numerical integration, interpolation, root finding, and minimisation.

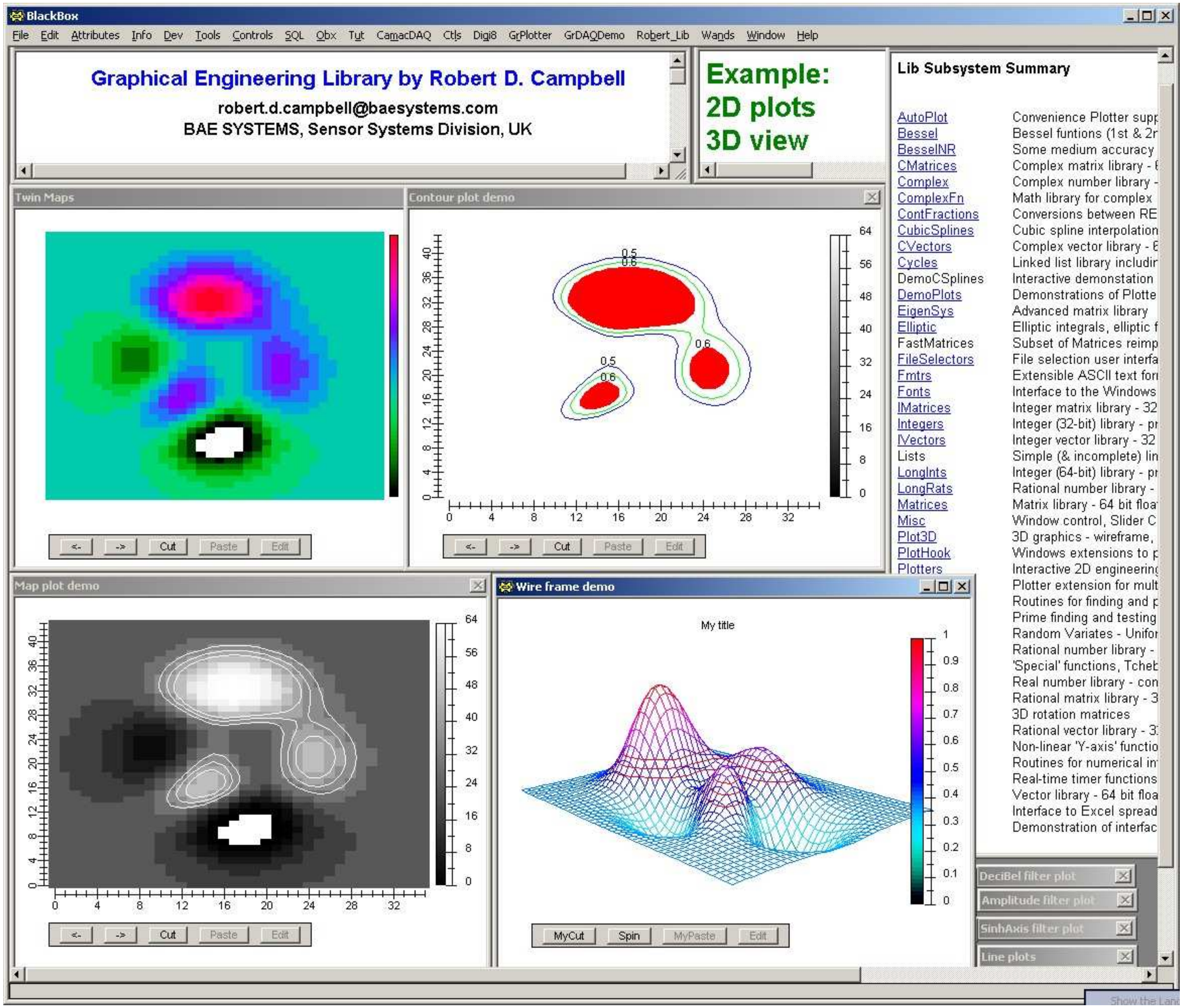
1D, 2D, and 3D plotting.

Includes interface to OpenGL.

Scientific plotting and computation library *Lib*

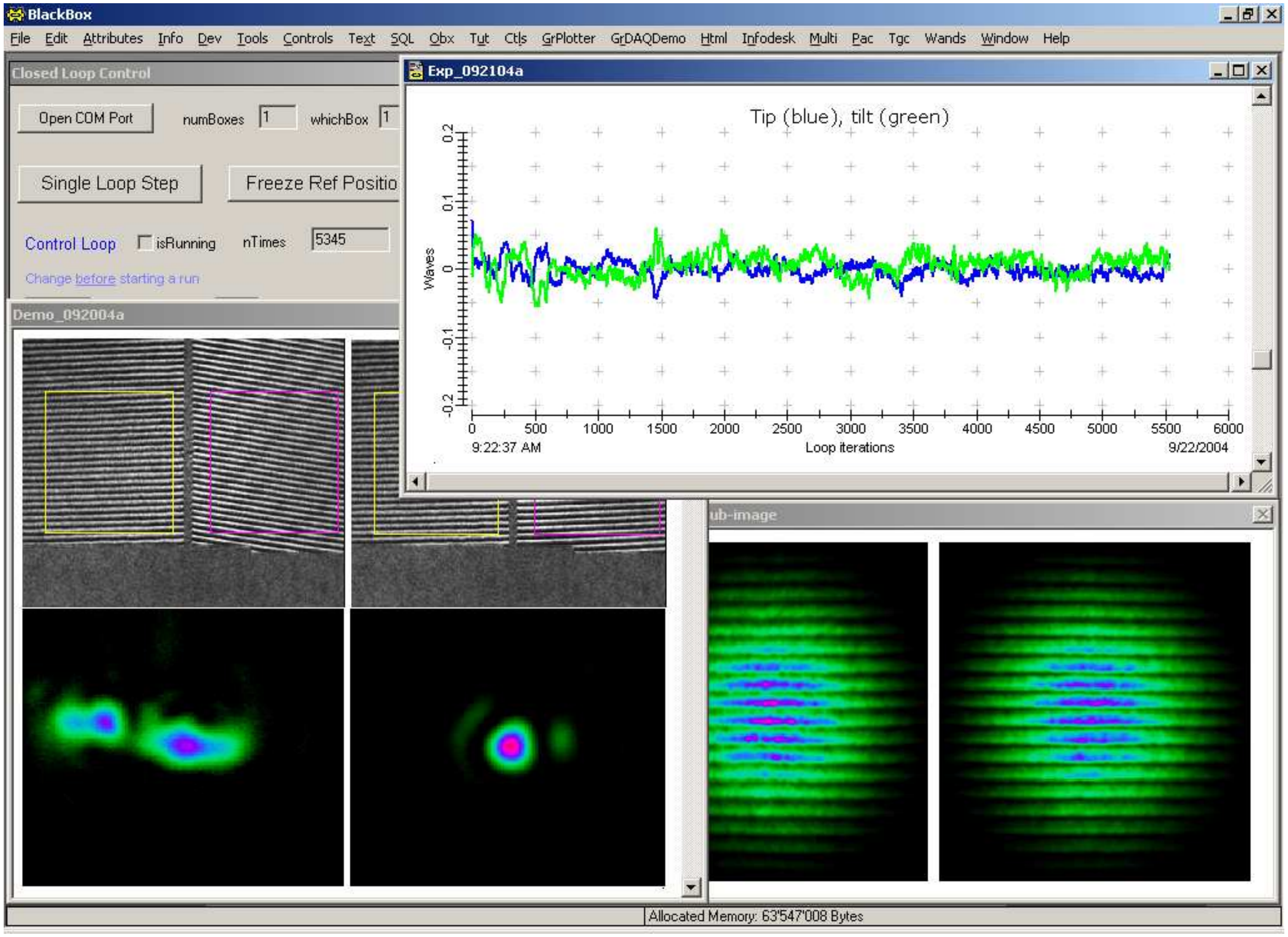


Scientific plotting and computation library *Lib*



A mission-critical application using *Lib*

Adaptive optics control package, including hardware control.



Graphical User Interface

GUI without pain.

GUI can be automatically generated.

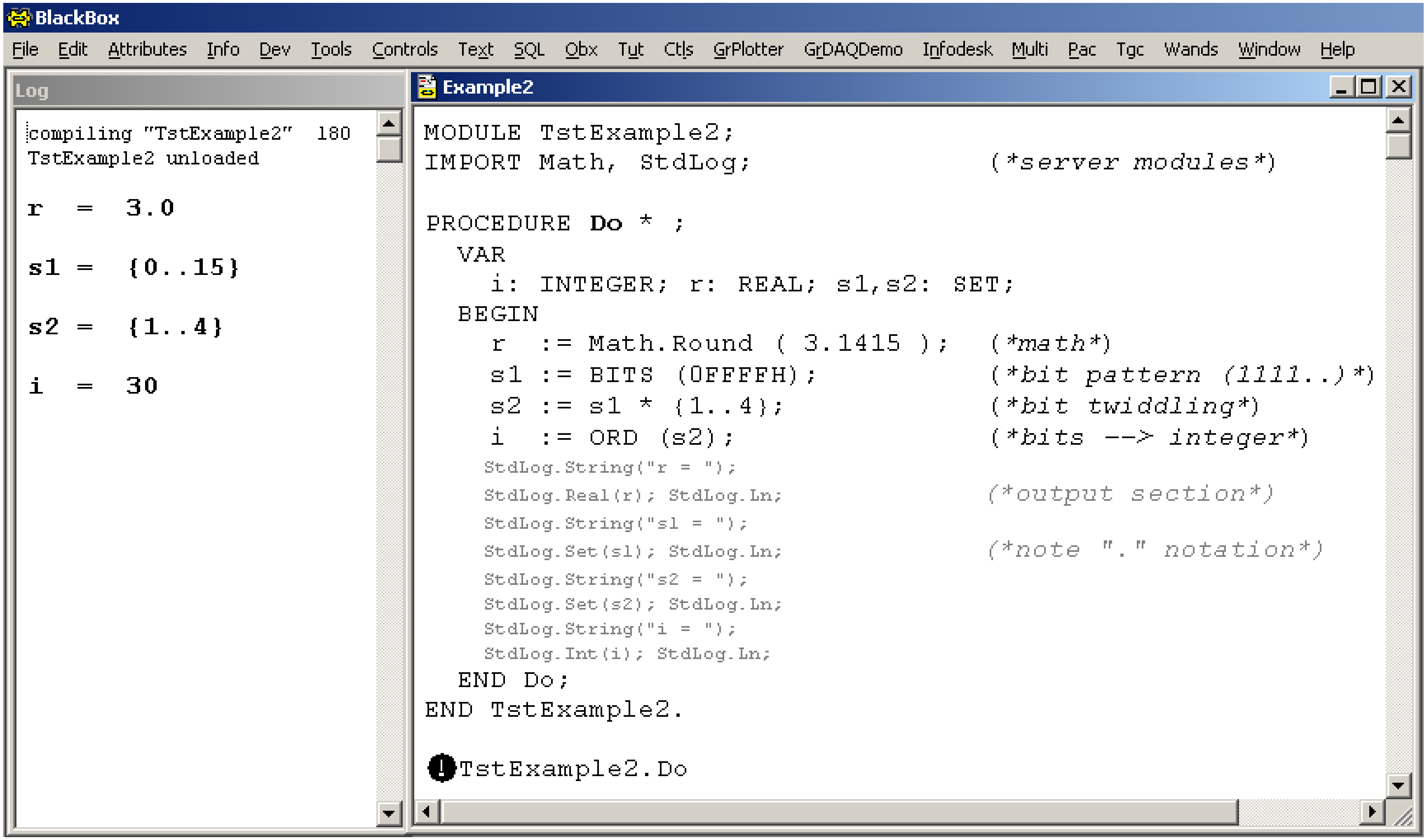
Before BlackBox, GUI design would take extensive effort.

Under BlackBox, GUI is not a problem anymore.

Work can focus solely on the algorithm.

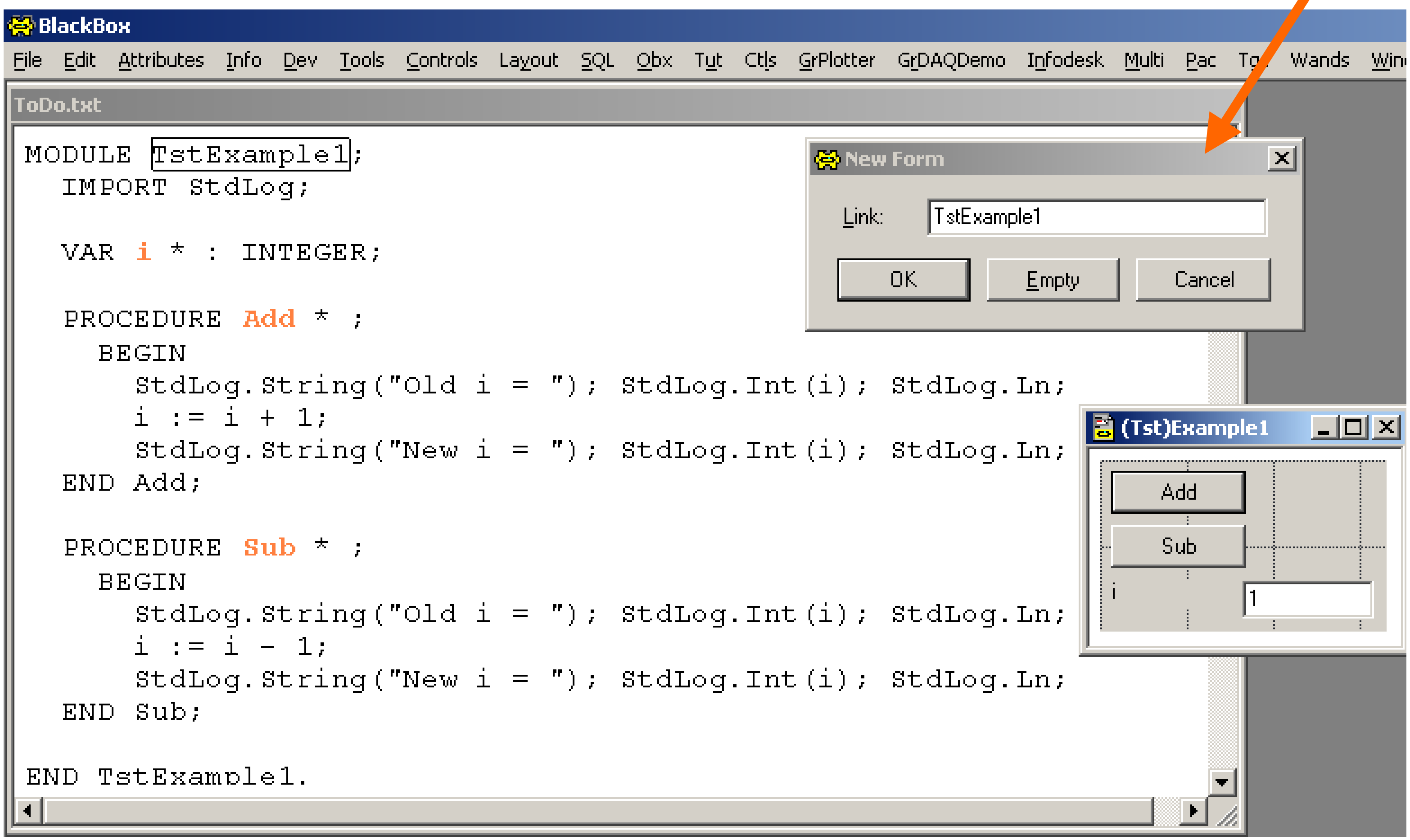
The developer's "GUI without GUI"

During development, command procedures can be called using "commanders".



Automatic GUI builder

- It is well known that usually most development effort is burnt on developing GUIs.
- Not under BlackBox! GUI development has been slashed with the automatic builder.



Automatic GUI builder

You do not have to think of the GUI too much.

- Write your procedures, declare the variables...
- … focus on the algorithm.
- Then press the button and you will get the GUI.

Other SW builds the GUI <u>code skeleton</u> that you have to fill in.

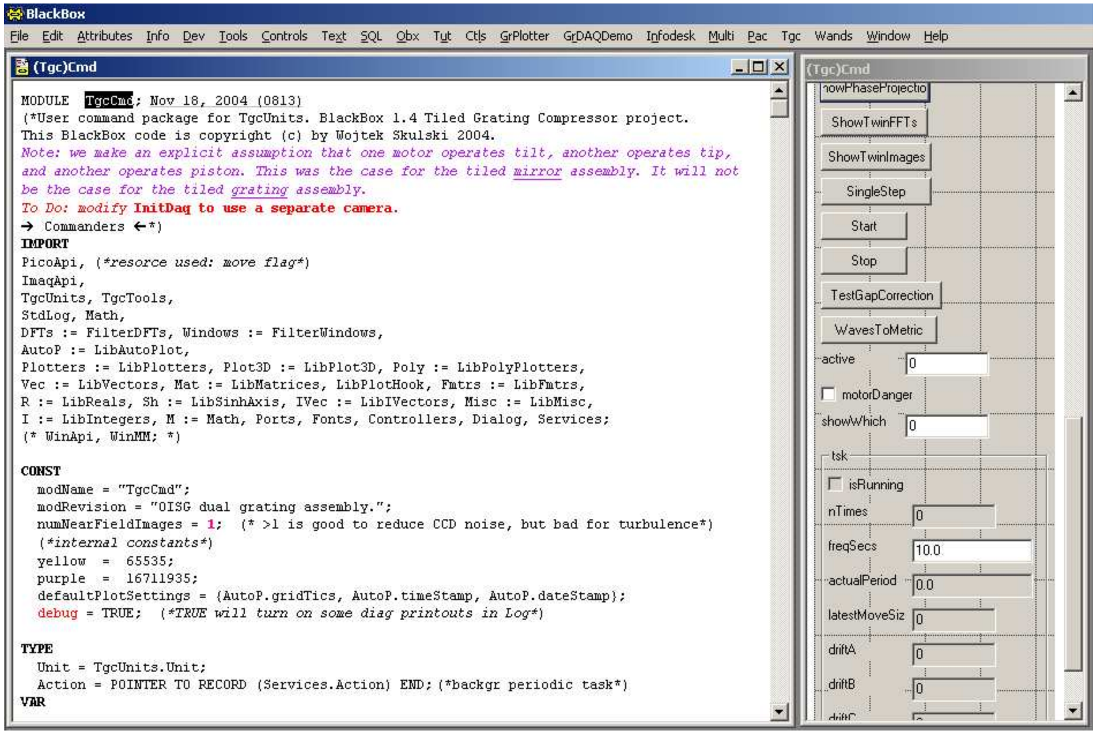
BlackBox does just the opposite.

- You write only the <u>application code</u>.
- GUI <u>panels</u> are automatically built.
- GUI code does not even exist.

Graphical User Interface panels

- Under BlackBox, the GUI panels do not contain any code.
- GUI can be automatically generated using built-in “metaprogramming” facility.
- After being generated, GUI panels can be customized by hand to suit the planned use.
- Several different GUIs can be put on top of the same software.
- GUI can be customized without rewriting the code.
- End-user GO-NOGO panels can coexist with the “expert GUI”. The code stays the same.

- Module TgcCmd and the auto-generated GUI displayed side-by-side. The GUI can be now rearranged by hand, some buttons can be deleted, other buttons can be added, and GUIs of other modules can be mixed with this one.
- It usually takes a very short time to design highly useful GUIs.
- All TGC panels have been created this way.



Interfacing with hardware

All HW interfacing is handled via DLLs.

DLLs are called by BlackBox.

Several HW projects have been completed with BlackBox at UofR.

CAMAC, USB, video frame-grabber card, mechanical actuators,

distance-measuring interferometer (DMI).

Steps in HW interfacing

- Collect information regarding your HW, example programs, and available DLLs.
- Read BlackBox Help --> Platform-Specific Issues.
- Install the DLL and its support software (if any) on your computer.
- Study example C programs and the .h files provided with the DLL.
  - C naming conventions are somewhat standardized.
  - `int, short,` or `long` are standard, but how about `DLLENTRY(BOOL)` ?
  - Such non-standard C names are #defined in the .h files provided by the DLL vendor.
- Write an interface module using "unmanaged" Pascal.
- Write an API module, a test module, and test GUI.
- Connect the HW and exercise the DLL using the test GUI.
- If you recompile the DLL interface module, you have to restart BlackBox.
- … or, use another instance of BB environment to test the DLL module.

"Unmanaged" Pascal is used to interface with DLLs

• Windows DLLs usually have no safety whatsoever. The DLLs that interface with HW are *kernel mode drivers* that can wreak havoc. Extreme care has to be used when passing parameters to the DLLs.

• Safety of C/C++ cannot be increased → Pascal safety has to be decreased.

• "Unmanaged" Pascal is compatible with C. (And can crash just as easily as C.)

• Locally decreased safety is marked with the keyword SYSTEM.

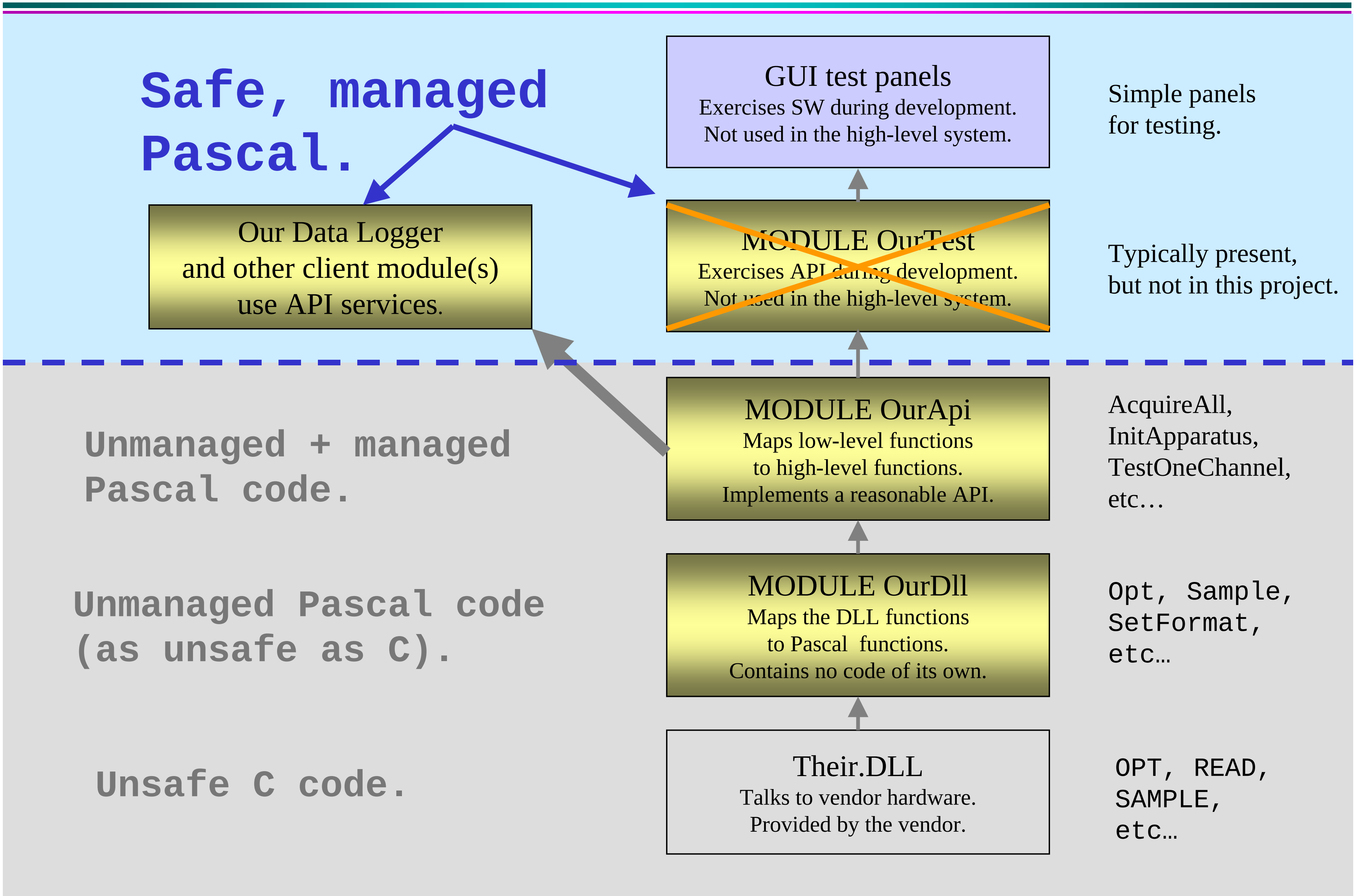
• C-compatible arrays and data structures are marked with [untagged].

• C-compatible pointers are formed with SYSTEM.VAL and SYSTEM.ADR.

• C-compatible pointers and data structures should be confined to low-level code.

• Never make use of unmanaged variables in high-level Pascal code.

Example: a complete subsystem



DLL interface module (functionally equivalent to .h file)

The interface module establishes “mapping” between DLL and Pascal functions .

Our Pascal module name

Their DLL name (without .DLL extension)

```
MODULE  OurDll  [“TheirDll"];

IMPORT SYSTEM;
```

“SYSTEM” is the “unmanaging” keyword

```
    (*This module can only contain DLL mappings, but no code.*)

    (*It is functionally equivalent to a C header file, i.e., .h *)

END OurDll.
```

Entering the domain of unmanaged Pascal code

The example OurAPI module contains both managed and unmanaged code.

"Unmanaging" keyword

Lower-level DLL interface module (prev slide)

Managed Pascal variables, public

Unmanaged C-compatible variables

```
MODULE  OurApi;

IMPORT SYSTEM, OurDLL;

VAR

    errorFlag  -  : BOOLEAN;

    msg        *  : ARRAY 128 OF CHAR


    errBuf   : OurDll.chrBufferType;    (*C-buffer for DLL data*)

    errPtr   : OurDll.chrBufferTypePtr; (*C-pointer to the above*)

END OurApi.
```

Example: mapping C-struct → Pascal

An arbitrary C-struct → corresponding unmanaged Pascal.

```
typedef struct SingleDataOutput {
    INT32   pos_data;       // position
    UINT32  time_data;      // time
    UINT32  vel_data;       // velocity
} SingleDataOutput;
```

Suppress Pascal type-safe information

```
TYPE
SingleDataOutputType *  = RECORD [untagged]
    pos_data * : INTEGER;
    tim_data * : INTEGER;
    vel_data * : INTEGER
END;
```

Pascal export mark

Examples: mapping C-arrays → Pascal

An arbitrary C-array → corresponding unmanaged Pascal.

Like all unmanaged code, requires keyword SYSTEM.

```
CONST bufLen = 512;

(* C-compatible array of 8-bit chars, null-terminated. Has to be
   allocated by the caller (either on stack or heap). *)

chrBufferType       *  = ARRAY  [untagged] bufLen OF SHORTCHAR;
chrBufferTypePtr    *  = POINTER TO chrBufferType;


(* C-compatible array of 16-bit integers. Has to be allocated by the
   caller (either on stack or heap). *)

int16BufferType     *  = ARRAY  [untagged] bufLen OF SHORTINT;
int16BufferTypePtr  *  = POINTER TO int16BufferType;
```

Example: DLL-function uses C-array → Pascal

DLL function uses C-array → corresponding <u>unmanaged</u> Pascal.

```
 DLLENTRY(INT16) OPT (char * ret_str)     // Original C-declaration

// Parameters of the DLL function:
//  - char * ret_str
//      - used to pass back some string
//
// Return INT16:      - number of characters in ret_str
```

Corresponding unmanaged Pascal in OurDll

```
PROCEDURE Opt * ["OPT"]                  (* Mapping C→Pascal in OurDll *)
    (
        infoPtr  : chrBufferTypePtr  (* Declared on previous slide *)

    ): Int16;                          (* Return value, 16-bit *)
```

… continued on next slide

Continued: DLL-function uses C-array → Pascal

Continued: how to call in Pascal the DLL function that uses C-array.

```
DLLENTRY(INT16) OPT (char * ret_str)     // Original C-declaration
```

```
PROCEDURE Opt * ["OPT"]                  (* Mapping C→Pascal in OurDll*)
    ( infoPtr  : chrBufferTypePtr        (* Declared on previous slide *)
    ): Int16;                            (* Return value *)
```

```
PROCEDURE InitApparatus *;               (* Calling Opt in OurApi*)
  VAR
    errBuf  : OurDll.chrBufferType;
    errPtr  : OurDll.chrBufferTypePtr;
BEGIN
  errPtr := SYSTEM.VAL(OurDll.chrBufferTypePtr, SYSTEM.ADR (errBuf));
  nchar  := OurDll.Opt ( errPtr );
```

Note: the line with VAL and ADR is an unmanaged C-like typecast, which is as dangerous as it would be in C.

Another example

Original C-declaration from TheirDll:

```
DLLENTRY(int) TheirInit(char *portname, unsigned int baudrate);
```

Mapping C → Pascal in the interface module OurDll:

```
TYPE PtrSTR = WinApi.PtrSTR;         (* C-pointer to a string of char*)

PROCEDURE TheirInit * ["TheirInit"] (* Assigning name C → Pascal*)
    (portname  : PtrSTR;            (* Null-terminated string*)
    baudrate  : INTEGER             (* Parameter passed by value*)
    ): INTEGER;                     (* Return value, 32-bit *)
```

Calling the Pascal function TheirInit in user Pascal code:

```
numBoxes := OurDll.TheirInit("COM1", 19200);
```

Example: interfacing with WinAPI

A complete set of Windows system calls is provided with BlackBox.

```
PROCEDURE ExampleApiCall (      ← typical Pascal declaration
    hDevice      :  HANDLE;     ← Windows 32-bit pointer
    controlCode  :  INTEGER;    ← ctrl code, hex
    buffer       :  PtrVoid;    ← buffer address
    bufferSize   :  INTEGER     ← buffer size
): BOOL;

result := WinApi.ExampleApiCall (                    ← calling in user code
  hDevice,                                           ← Windows pointer
  0220000H,                                          ← ctrl code, hex
  SYSTEM.VAL (PtrVoid, SYSTEM.ADR (buf) ),           ← buffer address
  SIZE ( bufType )                                   ← buffer size
);
```

… continues on the next slide

Example: interfacing with WinAPI, cont.

… continued.

Suppress Pascal type-safe information

```
TYPE bufType =  ARRAY [untagged] 100 OF BYTE;    ← C-compatible buffer
VAR  buf      : POINTER TO bufferType;


result := WinApi.ExampleApiCall (               ← example user call
    hDevice,                                     ← Windows 32-bit pointer
    0220000H,                                    ← ctrl code, hex
    SYSTEM.VAL (PtrVoid, SYSTEM.ADR (buf) ),     ← buffer address
    SIZE ( bufType )                             ← buffer size
);
```

Use either SYSTEM.VAL (WinApi.PtrVoid, buf), if the buf is a pointer to a heap object, or SYSTEM.ADR(buf), if buf is a stack object (i.e., not a pointer).

Example: parallel port interfacing

PC parallel port is the simplest HW we can play with.

Collect information:

Parallel Port Complete by Jan Axelson and her website www.lvr.com

Choose the most promising DLL. Install it on your computer.

Based on the above, I chose Inpout32.DLL from www.logix4u.com

Try to understand example C programs. (See next slide.)

Disentangling the examples was easy in this case, but in general it takes a while.

Write an interface module, see next slide.

Connect the HW and exercise the DLL.

Example: parallel port interface module

BlackBox

File Edit Attributes Info Dev Tools Controls Text SQL Obx Tut Ctls GrPlotter GrDAQDemo Infodesk Multi Pac Tgc Wands Window Help

C example

InpoutTest.cpp

```
// InpoutTest.cpp : Defines the entry point for the console applicat
//

#include "stdafx.h"
#include "stdio.h"
#include "string.h"
#include "stdlib.h"
/* ----Prototypes of Inp and Outp--- */

short _stdcall Inp32  (short PortAddress);
void  _stdcall Out32  (short PortAddress, short data);

/*--------------------------------*/

int main(int argc, char* argv[])
{

  int data;

  if(argc<3)
  {
    //too few command line arguments, show usage
    printf("Error : too few arguments\n\n***** Usage *****\n\nInpout
\nInpoutTest write <ADDRESS> <DATA>\n\n\n\n\n");
  }
  else if(!strcmp(argv[1],"read"))
  {

    data = Inp32(atoi(argv[2]));

    printf("Data read from address %s is %d \n\n\n\n",argv[2],data);

  }
  else if(!strcmp(argv[1],"write"))
  {
    if(argc<4)
    {
      printf("Error in arguments supplied");
      printf("\n***** Usage *****\n\nInpoutTest read <ADDRESS> \nor
<ADDRESS> <DATA>\n\n\n\n\n");
    }
    else
    {
    Out32(atoi(argv[2]),atoi(argv[3]));
    printf("data written to %s\n\n\n",argv[2]);
    }
```

BB module

Inpout32DLL

```
(*BlackBox interface module for inpout32.dll. (c) Wojtek Skulski 2004
C source code from www.logix4u.com. Download and unpack
inpout32_source_and_bins.zip
The directory "test applications" will contain C apps to test the DLL.
  The following functions are defined by the DLL.
  short   _stdcall  Inp32 (short PortAddress);
  void    _stdcall  Out32 (short PortAddress, short data); *)

MODULE TstInpout32Dll ["inpout32"] ;
IMPORT SYSTEM;
CONST
  modName * = "TstInpout32";       (*Module identification*)
  modRevision * = "BlackBox HW interfacing demo.";
TYPE
  short* = SHORTINT;       (* see BlackBox Help -> Platform-Specific Issues *)

(* --------------------------------------------------------------------------
  Inp32   get one byte of data from the parallel port.
  The data item is 16 bits, but we should use only 8 bits.
 --------------------------------------------------------------------------*)

PROCEDURE Inp32 * ["Inp32"]
  (
    addr : short      (* HW address of the parallel port*)
  )      : short;     (* returns 16-bit signed int, use only lower 8 bits*)

(* --------------------------------------------------------------------------
  Out32   send one byte of data to the parallel port.
  The data item is 16 bits, but we should use only 8 bits.
 --------------------------------------------------------------------------*)

PROCEDURE Out32 * ["Out32"]
  (
    addr : short;        (* HW address of the parallel port*)
    dta  : short         (* data item, 16 bits, use only lower 8 bits *)
  );

END TstInpout32Dll.
```