GENERAL STATUS OF ROOT GUI

I. Antcheva, B. Bellenot, R. Brun, F. Rademakers, CERN, Geneva, Switzerland
V. Onuchin, IHEP, Protvino, Russia

Abstract

ROOT [1, 2] as a scientific data analysis framework provides a large selection of data presentation objects and utilities. The graphical capabilities of ROOT range from 2D primitives to various plots, histograms, and 3D graphical objects. The object-oriented design of ROOT offers considerable benefits for developing object-oriented user interfaces.

The basic features and the progress made with the recent user interface developments in ROOT are presented in this paper.

BASIC FEATURES

The ROOT Graphical User Interface (GUI) classes support an extensive and rich set of user interface elements with a common look and feel. A GUI element (known also as a widget) displays information or provides a specific way for users to interact with the application or the operating system. Widgets include icons, pull-down menus, buttons, progress bars, scroll bars, list trees, toggle buttons, and many other devices for displaying information and for inviting, accepting, and responding to user actions.

The object-oriented GUI library of ROOT offers a full set of application design utilities to programmers. A rich and complete set of widgets can be arranged into container frames that use ROOT container classes for fast look up.

Cross-Platform User Interface

The ROOT framework offers considerable benefits for developing a fully cross platform object-oriented user interface.

The widget classes interface to the platform-dependent low level graphics system, via the abstract class *TVirtualX*. Concrete versions of this abstract class have been implemented for X11, Win32, and Qt [3]. Porting the user interface to any new platform only requires the new implementation of *TVirtualX*.

The benefit of applications running on more than one kind of computer is obvious - it increases the program's robustness, makes maintenance easier and improves the reusability of the code.

Signals/Slots Communication Mechanism

The object-oriented, event-driven programming model supports the modern signals/slots communication mechanism as an advanced object communication concept. It largely replaces the old call-back functions for handling actions in GUIs. Signals and slots are just like any object-oriented method implemented in C++.

The signals/slots communication mechanism is integrated into the ROOT core. It uses dictionary information and the CINT interpreter to connect signal methods to slot methods. It facilitates programming since it allows a total independence of the interacting classes.

On interactions, widgets send out various *signals (event senders).* Public object methods can be used as *slots (event receivers).* Signals and slots can be connected together enabling different components to snap simultaneously like Lego blocks, leaving possibilities for adding new pieces in the future.

Saving User Interface in a Macro

Any ROOT GUI class has *SavePrimitive* method that allows a C++ code generation in a macro file. Users can invoke those methods and save any application layout or any dialog layout by using *Ctrl+S* short cut. The generated macro can be modified and then executed via the CINT interpreter.

Executing the macro restores the complete original application layout and emulates the action handling based on all existing signals/slots connections in a global way.

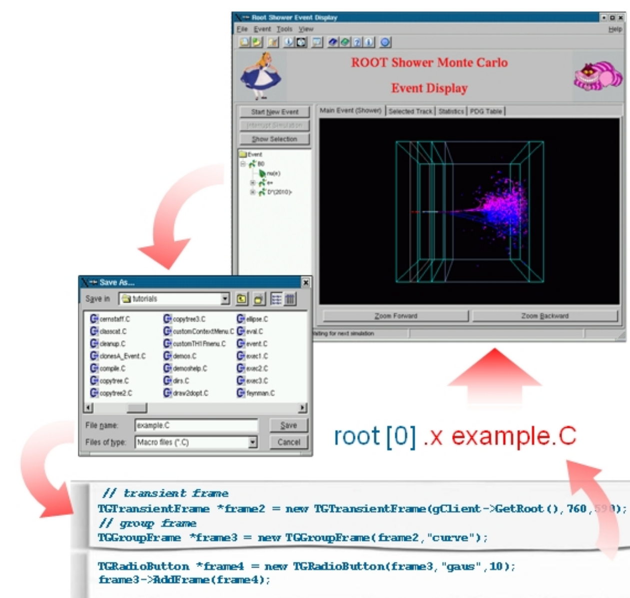

Figure 1: Generating the GUI code

CANVAS INTERFACE

The canvas window comprises the following user interface elements around the ROOT canvas: menu, tool and status bars, and the editor frame.

The *Menu bar* provides access to common and frequently used actions such as operations with files, print, inspecting objects, and toggling different interface elements, etc. The dockable *tool bar* provides shortcuts for several menu items and buttons for primitive drawing in the canvas.
The *Canvas* is the working area used for object visualization. The *Status bar* displays descriptive messages about the selected object.

If the *Editor frame* is activated, it responds dynamically and presents the user interface according to the selected object in the canvas.

The *Style Manager* can be activated from the canvas Edit menu. It handles general graphics, settings in the ROOT session and allows style editing.

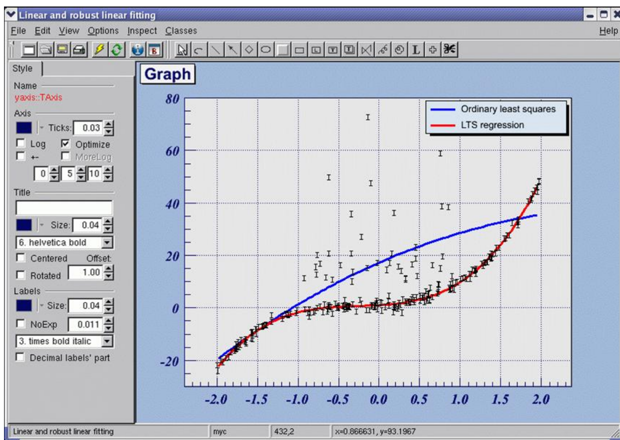

Figure 2: The user interface of ROOT canvas

NEW AND IMPROVED WIDGETS

The ROOT GUI project has made progress in several new developments of user interface components for a standard application environment.

The new *dockable frame widget* allows undocking or docking of menus, tool or status bars, or the collapsing of these bars from the application window.

Thanks to the *Multiple Document Interface* (*MDI*) widgets, it is possible now to have an application with a main window and various numbers of child windows, which are shown inside.

The *triple slider widget* provides an easy selection of a range and a value. The pointer position, defining the value, can be constrained into a selected range or can be set relative to it.

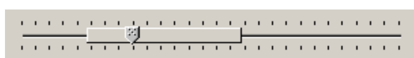

Figure 3: The triple slider widget

Since the new concept of '*Pseudo-windows*' was implemented in the container classes *TGListView*, *TGListBox*, *TGListTree*, and *TGComboBox*, it is now possible to draw and scroll a huge amount of items in list views, list boxes, etc.

The possibility of turning on/off pieces of the tree hierarchy on the list tree nodes was implemented in the ROOT object browser. This important feature is available for use when browsing detector geometries. This new feature is used in the ROOT object browser to toggle the visibility of geometries.

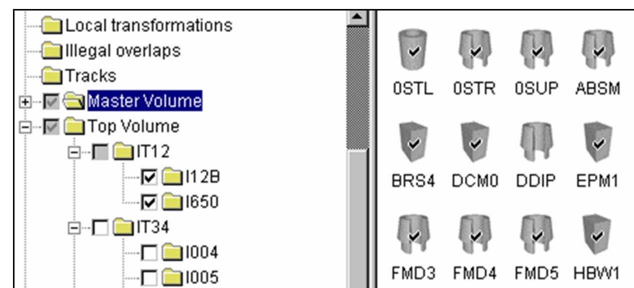

Figure 4: Check boxes in list tree nodes

GRAPHICS EDITOR

The ROOT graphics editor [4] is split into discrete units of so-called object editors. Any object editor provides an object specific user interface that shows up when the corresponding object is selected. This interface design is built with a capacity for growth and can be extended easily by user-defined object editors.

The graphics editor is modular. It loads the corresponding editor *<class>Editor* according to the *<class>* of the selected object in the canvas window, e.g. if the user selects a *TAxis* object, the user interface provided by *TAxisEditor* is activated and ready for use.

Importantly the rules (outlined below), describe the path to follow when creating your own object editor. Designed in this way, it will be recognized and loaded by the graphics editor of ROOT. The rules are:

(a) Derive the code of your object editor from the base editor class *TGedFrame*.

(b) Keep the correct naming convention: the name of the object editor should be the object class name concatenated with the word 'Editor'.

(c) Register the new object editor into the list *TClass::fClassEditors* in its constructor.

(d) Use the signals/slots communication mechanism for event processing.

(e) Implement as a slot, the method *SetModel* where all widgets are set with the current object's attributes. This method is called when the graphics editor receives a signal from the canvas saying that an object has been selected.

(f) Implement all necessary slots and connect them to appropriate signals that GUI widgets send out. The GUI classes in ROOT are developed to emit signals, whenever they change a state in which others might be interested.

As we noted already, the signals/slots communication mechanism allows total independence of the interacting classes.

STYLE MANAGER

This new Graphical User Interface is created to manage different styles in a ROOT session. It allows users to import a style from a canvas or a macro, to select a style for editing, to export it in a C++ macro, to apply a currently selected style on a selected object in a canvas or on all canvases, to set it as the *gStyle*.

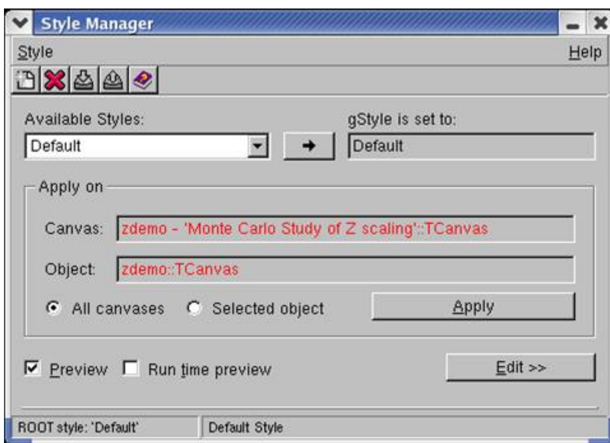

Figure 5: The main window of ROOT Style Manager

The interface is composed of two parts:

- The top level interface manages a list of all available styles for the current ROOT session and shows the currently selected one;
- The style editor deals with the settings of the currently selected style. It becomes active and shows up after users click on the 'Edit >>' button.

A preview of the selected canvas helps for precision work. It can be updated dynamically at run-time or by request to show how the edited style looks. All changes made in the style editor can be cancelled and the edited style can be restored to the last saved state in a macro.

GUI BUILDER

The GUI builder can greatly simplify the process of designing GUIs based on ROOT widget classes.

Its purpose is to provide a complete tool that allows ROOT users to build basic applications with menus and dialogs.

The user interfaces can be assembled by dragging GUI elements from the widgets' palette and dropping them in the design area that represents a main application window or a dialog window.

Widgets can be resized, grouped and moved to desired positions, and can be easily assigned to one or another parent widget. Each widget's properties can be set using the Property Editor tools, of the builder.

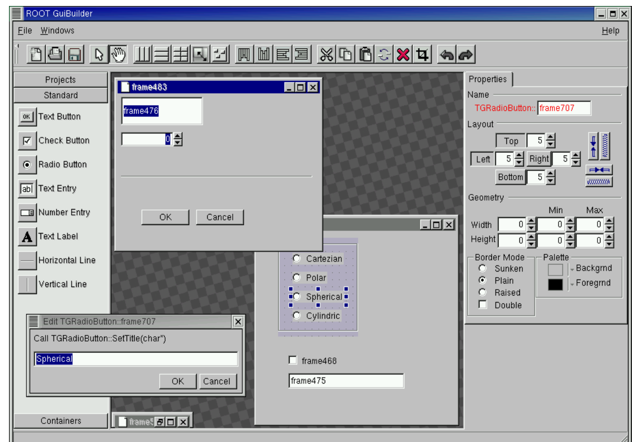

Figure 6: The ROOT GUI builder

By using *Ctrl+S* short keys or *SaveAs* file dialog, users can generate C++ code in a macro that can be edited and executed via the CINT interpreter:

```
root[0] .x example.C
```

ROOT AND QT

The different ROOT/Qt integration interfaces give Qt users easy access to the ROOT graphics in their Qt-based user interfaces.

Qt BNL

Qt-BNL [5] implementation of *TVirtualX* (Qt-layer) works as a ROOT plug-in shared library. Users can turn on/off the Qt interface layer at 'run time' with no need for changing or recompiling their applications including both: Qt and ROOT features.

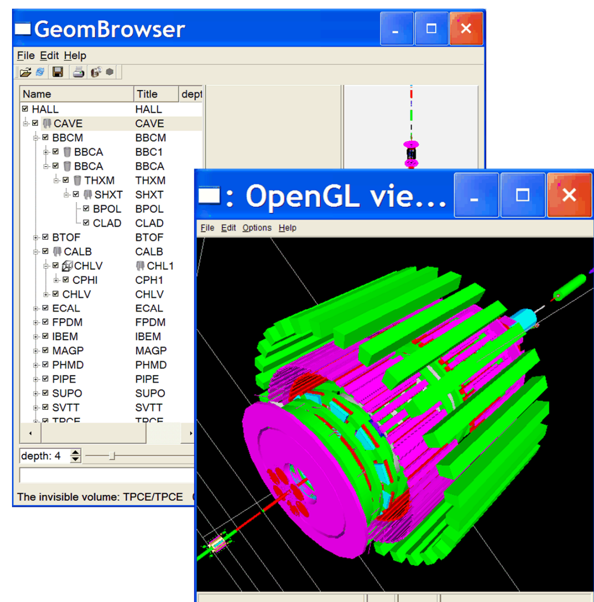

Figure 7: The QtBNL interface example

The primary goal of Qt BNL is to allow “embedding” the ROOT canvas into the arbitrary Qt widgets and using it with other Qt-based components, as well as with Qt-based third party libraries. The *TGQt* class in ROOT is the basic interface for the Qt GUI system and provides the concrete implementation of the abstract *TVirtualX* class. Thanks to it, as soon as users create a *TApplication* object, they can start using Qt classes as well.

The Qt BNL interface is planned to provide complete compatibility with ROOT GUI classes.

Qt GSI

This lightweight interface [6] has been available since 2001 on Linux and makes possible the use of Qt and ROOT together. Its Windows port is currently under development.

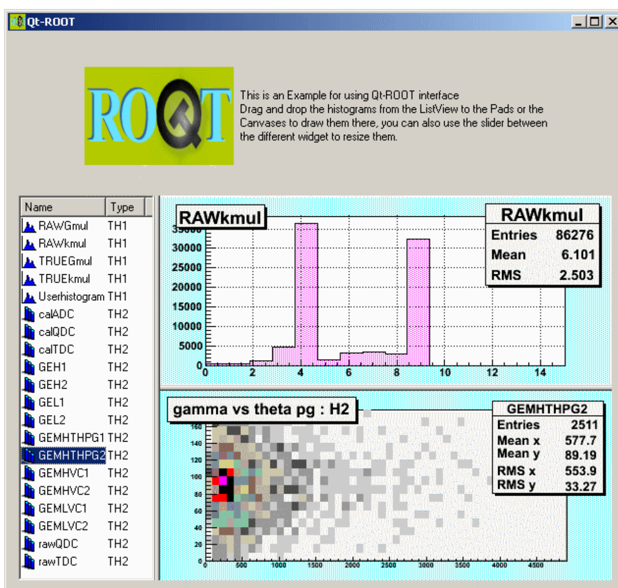

Figure 7: The QtGSI interface example

The Qt GSI interface allows the use of KDE/Qt based libraries for writing ROOT based applications. It follows the general design of native ROOT GUI classes and provides corresponding Qt based interface classes.

The Qt internal event loop, is used to drive the Qt application user interfaces; at the same time the ROOT event loop handles all ROOT events for native ROOT GUIs, timers, signals, i.e. as a result Qt widgets are rendered via Qt, while ROOT widgets are rendered either via *TGX11* or *TGWin32GDK*.

CONCLUSION

The ROOT GUI is a rich and powerful, scriptable and cross-platform library. It provides a solid basis for the development of many additional widgets and user interface components, like a Fit panel, Help browser, object editors and event displays. The GUI builder’s function is to make the task of designing complex user interfaces much easier.

REFERENCES

[1] R. Brun, F. Rademakers, ROOT - An Object Oriented Data Analysis Framework, Linux Journal, 1998, Issue 51, Metro Link Inc.

[2] http://root.cern.ch/

[3] TrollTech, Qt, http://www.trolltech.com

[4] I. Antcheva, R. Brun, C. Hof, F. Rademakers, The graphics editor in ROOT, NIMA 559 (2006), 17-21

[5] V. Fine, http://root.bnl.gov

[6] D.Bertini, M.Al-Turany. QtRoot. http://www-linux.gsi.de/~go4/qtroot/html/qtroot.html