Raw-data Display and Visual Reconstruction Validation in ALICE

Matevž Tadel
CERN, Switzerland
E-mail: matevz.tadel@cern.ch

Abstract.
ALICE Event Visualization Environment (AliEVE) is based on ROOT and its GUI, 2D & 3D graphics classes. A small application kernel provides for registration and management of visualization objects. CINT scripts are used as an extensible mechanism for data extraction, selection and processing as well as for steering of frequent event-related tasks. AliEVE is used for event visualization in offline and high-level trigger frameworks. Mechanisms and base-classes provided for visual representation of raw-data for different detector-types are described. Common infrastructure for thresholding and color-coding of signal/time information, placement of detector-modules in various 2D/3D layouts and for user-interaction with displayed data is presented. Methods for visualization of raw-data on different levels of detail are discussed as they are expected to play an important role during early detector operation with poorly understood detector calibration, occupancy and noise-levels. Since September 2006 ALICE applies a regular visual-scanning procedure to simulated proton-proton data to detect any shortcomings in cluster finding, tracking and primary & secondary vertex reconstruction. A high-level of interactivity is required to allow in-depth exploration of event-structure. Navigation back to simulation records is supported for debugging purposes. Standard 2D projections and transformations are available for clusters, tracks and simplified detector geometry.

1. Introduction

AliROOT, the offline environment of the ALICE experiment [1], is built on top of the ROOT framework [2]. The only additional external dependencies of AliROOT are the Monte-Carlo transport engines Geant3, FLUKA and Geant4 (in order of inclusion). This simplifies not only the build system and installation but also makes porting to new platforms much more viable. Event-data is stored in a set of ROOT files, each containing a specific data-type for given sub-detector (e.g. a file named `TPC.Digits.root` contains time-projection chamber digits). Internally, all data is stored in ROOT trees and clones-arrays are used for logical grouping, e.g. to merge all hits of a given primary particle or all digits of a given detector module.

ALICE computing is visibly marked by the large data-volume of lead–lead events, produced by around 60 k primary particles entering the sensitive volumes and producing 1.5 GBytes of data during a full simulation/reconstruction pass, including 600 k simulated particles (1.5 M actually tracked), 150 M TPC hits, 3.2 M TPC clusters and 16 k reconstructed tracks. The raw-data size is 80 MBytes and typical size of event-summary data is 4 MBytes. The large data-volume has many consequences also for the visualization framework as versatile selection algorithms must be provided to limit the displayed data to any particular range or region of interest to the user.

Development of ALICE Event Visualization Environment (AliEVE) started in 2005 with prototyping of the visualization elements and selection algorithms within the GLED framework [3]. In 2006 the the first prototype using the ROOT graphical user-interface and 3D graphics was assembled [4] and since then the system has been growing in scope and versatility. OpenGL support in ROOT has also been augmented as a part of this development [5].

2. AliEVE architecture

The architecture of AliEVE is heavily influenced by the format and quantity of the input data, variability of its desired visual representations and by the set of selection and transformation algorithms that need to be supported on various stages of the data processing. The requirements for supported input-data types are the following:

(i) detector geometry, all types of simulation & reconstruction data, including reconstructed physics-objects;

(ii) raw-data and digits;

(iii) high-level trigger events and performance monitoring of trigger-algorithms;

(iv) on-line detector monitoring data.

Due to the mutual incompatibility of the input sources a clear separation must be made between input-processing, object management and data-display components of the framework. It is also desirable that the functionality is introduced gradually: simple visualization tasks should only make use of the basic framework elements while advanced mechanisms must provide enough flexibility to allow construction of applications with complex internal state and heavily cross-referenced data-structures.

Structural aspects of the architecture are also determined by a set of implementation constraints arising from AliROOT development environment and work-flow organization. Further, restrictions on the manpower allocated for the core development and short required time-to-delivery,[1] forced us to adopt the extreme-programming development practices with aggressive core API prototyping and a wide spectrum of test cases and demonstration programs. The main implications for the architectural structure are summarized in the following points.

- Instead of building a monolithic application provide an extensible framework where user-code can be plugged into practically every level of data-processing.
- ROOT is the only external software dependency. This implies usage of native ROOT GUI widgets and ROOT interface to OpenGL graphics.
- Adopt the same component philosophy as ROOT: provide a modular, loosely coupled toolbox of classes with sample, default implementations of the higher-level functionality. Use CINT scripts for application configuration and for steering of the input-data processing.
- Introduce new elements into the framework as they are needed on all levels of the system, including ROOT.

It was recognized from the very beginning that a clear separation between the general, experiment independent part of the framework and the ALICE specific functionality is necessary. First, it improves the the overall modularity of the system and allows many visualization tasks to be performed independently of AliROOT.[2] Second, this allows other experiments to directly reuse the stand-alone ROOT-based visualization framework or to use it as a base for more complex development. The two modules are described in the following sub-sections.

[1] The development team consists of two developers, a project-lead & a high-level junior. 20 months were available for the development after the early-prototype release.

[2] All ALICE data is stored in a set of ROOT trees that can be, to some degree, accessed without the source class definitions. Alternatively, ALICE libraries can be loaded on demand.

2.1. REVE – ROOT Event Visualization Environment

REVE is the experiment independent part of the ALICE visualization environment and depends only on ROOT. It is currently in the process of becoming a ROOT module and will thus be released from its cradle within AliROOT. Both ATLAS and CMS have expressed interest in exploring it as a light-weight, complementary alternative to their visualization frameworks.

The components of REVE can be put into three main categories, based on the role they play within the framework.

(i) **Application core** consists of general services exposed to users via an instance of the `Reve::Application` class and some global functions of the `Reve` namespace. Together, they provide four main functionality units:
- management of object-browsers, 3D scenes and 3D viewers;
- registration of visualization objects;
- event management & navigation;[3]
- execution environment for CINT scripts.

(ii) **Framework base-classes** implement the low-level functionality of visualization and GUI objects that is used by the application core to perform object and state management as well as to provide a reasonable level of feedback to the user (e.g. object names and titles, color-marking, object inspection via GUI and command-line interfaces).

(iii) **Basic 3D visualization classes** serve as initial building blocks for simple visualization tasks, as base-classes for more advanced visualization classes or simply as examples of framework usage. The standard HEP visualization classes (geometry, points and tracks) are discussed in Sec.3 and base-classes for raw-data visualization in Sec.4.
All classes in this category are equipped with accompanying GUI editors and OpenGL rendering classes. Following the ROOT's naming convention, the `Track` class has accompanying GUI implemented in class `TrackEditor` and GL renderer in class `TrackGL`. For details of this mechanisms see [5] and [6].

A top-level window of REVE as a standalone application is shown in Fig.1.

2.2. AliEVE – ALICE Event Visualization Environment

AliEVE encompasses code that is specific to ALICE and requires the presence of AliROOT libraries. At the time of writing, it provides access to ALICE data, visualization of raw-data & detector modules and `C++` scripts that produce the visualization objects for all other data-types.

(i) **Access to ALICE data.** Visualization code accesses data in a random fashion, based on user input and not on any predetermined pattern as is the case during simulation or reconstruction. Thus we need to shield the AliROOT event-loading functionality from the visualization data-consumers to prevent multiple loading of the same data and to simplify the user interface by covering the most frequent usage patterns. Additionally, it must support loading of detector geometry, magnetic filed maps, alignment-data and detector-conditions database.
Parts of the event-navigation could in principle be solved on a general level but its implementation is currently still kept in AliEVE. It is possible to register a set of commands that are executed after the loading of a new event.

[3] Two default containers for visualization objects are provided: one for global objects (like geometry) that remain resident during event-navigation and another one for event-based data that needs to be dropped. Due to the variability of input-data only a very basic infrastructure can be provided for this task.

(ii) **Raw-data and detector-module visualization** needs to be treated with special care as it requires direct access to raw-data reading functionality as well as to the specifics of detector structure and read-out electronics such as module positioning, segmentation and channel numbering conventions. The most advanced solution is required for TPC, in part also due to its large data-volume. Other complex detectors (e.g. ITS, TRD, TOF) extend REVE base-classes for raw-data representation, mostly to provide tools for user-interaction. For simple detectors with small data-volume and little segmentation (e.g. VZERO, T0), the visualization is provided by scripts that use REVE classes directly.

(iii) **Visualization scripts** are CINT macros that perform the actual data extraction, create and fill the visual representation objects and register them into the application. In a sense, they provide a bridge between the ALICE data and the visualization structures and relieve the core application of any knowledge about AliROOT internals (other than event-data interface). The default demonstration scripts are provided with the AliEVE distribution and are named by sub-detector and data-type, e.g. `tpc_digits.C`, `trd_clusters.C`, etc.

Every effort is made to keep AliEVE as small and as simple as possible. Most of this is already achieved by pushing all the general functionality into the REVE base-classes. Another simplification comes from the usage of CINT scripts for data-extraction steering. By providing a concise interface for their invocation and exception-throwing methods for obtaining handles into the ALICE data, their framework induced overhead is reduced to a bare minimum. Further, as standard ALICE data-containers are returned by these functions, the macros retain the look and feel of standard AliROOT code. This helps users first to understand the macros and second to tailor or enhance them for their specific needs without any further complications. A screen-shot of AliEVE in action is presented in Fig.2.

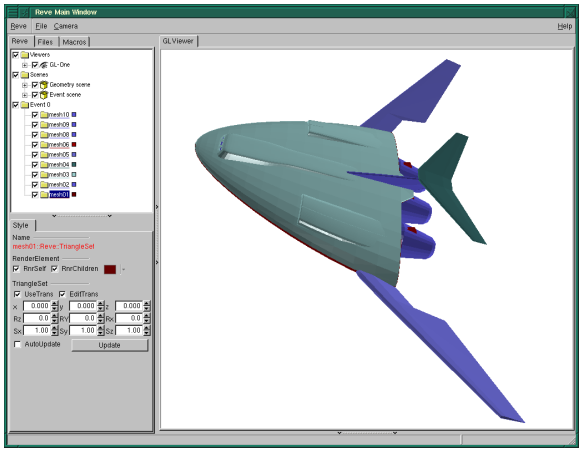

Figure 1. Standard REVE window showing an imported 3D-studio model (B. Bellentot).

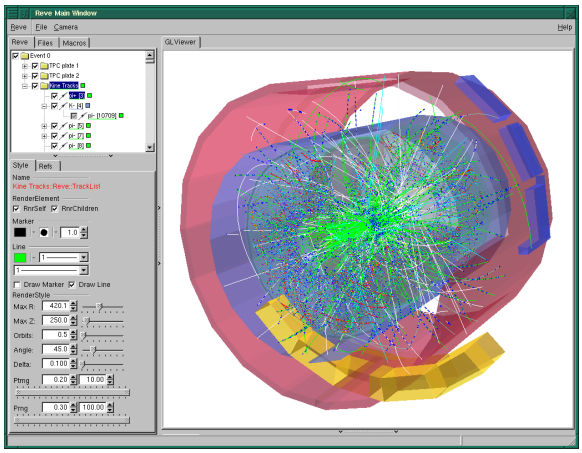

Figure 2. An ALICE p–p@14 TeV event showing simplified geometry and kinematics.

3. Standard visualization classes

In this category we describe the traditional event-visualization classes of REVE that are present in practically any event-display program. These programs are typically used by physicists, to gain insight into the structure of the detector and the topology of events, and by developers of the experiment software-frameworks for visual debugging of simulation and reconstruction algorithms. All these classes are also used for visual reconstruction validation of ALICE (Sec.5).

3.1. RenderElement – the visualization base-class
`RenderElement` is the base-class of all visualization classes in REVE. It provides interfaces between the objects and the application core, the 3D rendering system and GUI. Each render-element can have an arbitrary number of children and also holds lists of its parents and GUI representations so that the update requests can be propagated properly. All elements are reference counted and by default auto-destructible.

3.2. Geometry
ROOT includes a native geometrical modeler, TGeo, that provides methods for construction of detector geometries, particle tracking and volume visualization via the `TGeoPainter` class [7]. REVE supports two methods for geometry visualization: the first one uses TGeo directly and the second one presents pre-extracted volume-shape tessellations.

3.2.1. Display of full TGeo geometries REVE allows simultaneous display of several independent geometry sub-trees, possibly belonging to different geometries, via a wrapper-class `GeoTopNodeRnrEl`. It encapsulates a reference to geometry-manager and to the top-node to be displayed as well as the visualization parameters supported by `TGeoPainter` including the depth of geometry-tree traversal. During actual painting, the necessary global variables are set-up and then the control is passed to the `TGeoPainter`.

The `GeoNodeRnrEl` class allows further link between TGeo and REVE by providing representation of children nodes in the REVE object browser. Users can select individually which nodes to draw, block the descent of the painting algorithms from a given node and change node and volume colors.

3.2.2. Display of extracted shape-data In event-display applications one is usually not interested in the details provided by the full geometry description, especially as it includes all the support structures down to the minute details. Instead, one prefers to see a carefully selected set of relevant sensitive volumes or even just the envelopes of whole sub-detector systems. Such selection can be stored as a hierarchy of `GeoShapeRnrEl` objects that incorporate a `TGeoShape` data, its global transformation matrix, color and visibility flags. It can be stored in an independent ROOT file and typically requires only 1% of the space required for the full geometry.

3.3. Hits & clusters
Hits and clusters can be visualized by using the `PointSet` class, holding an array of 3D points that can be rendered with various marker styles and colors. To allow for backward-navigation, a reference to an external object (via ROOT's `TRef` class) can be specified for each point (optionally a `PointSet` can own the reference objects and delete them upon its destruction).

For filling of the data one can use a sequential method, specifying coordinates and an external reference for each point in turn, or use a special `TPointSelector` class that invokes the full ROOT machinery for selecting data from `TTree`'s.

The `PointSetArray` class implements an interactive 3D-histogram by encompassing an array of `PointSets`'s and providing a special filling method that allows a user to ascribe an additional value to each point, like deposited charge for a hit or sum of signals for a cluster. After that the user can interactively select the range of that value and thus control which subsets are actually displayed. Currently a single additional parameter is supported.

3.4. Trajectories, particles & tracks

The `Track` class can be used to represent particle trajectories and supports extrapolation and interpolation in a constant magnetic field.[4] An arbitrary number of control-points can be specified along the track, to mark one of: a) position/momentum reference, b) daughter creation point, or c) decay point. General track rendering parameters (e.g. maximum extrapolation radius and z-coordinate, required precision, etc) are stored in a separate class `TrackRnrStyle`. Usually all tracks from a given data-source reference the same render-style object thus allowing the general parameters to be edited for all of them simultaneously.

Tracks can be put into a hierarchical structure (as required for display of kinematics) or combined to represent composite reconstructed physics objects like V0's, kinks and resonances. A collection of tracks can be put into a `TrackList` object that provides control over common track rendering parameters and interactive selection of displayed momentum ranges.

4. Raw-data visualization

Visualization of raw-data is, in comparison to hits or clusters, complicated by the implicit digit positioning based on the module and channel number. Further, a signal value must always be shown in some fashion, usually by color or size of the digit's visual representation. In this section the REVE base-classes for raw-data presentation are discussed first. Special sub-sections are devoted to a more specific solutions used for display of TPC and ITS raw-data.

4.1. REVE base-classes

4.1.1. Support classes The support classes encapsulate functionality that is shared among several visualization classes and further, by several instances of a given visualization class, implying that they must be referenced via pointers from the visualization objects. These objets are reference-counted with automatic destruction, thus relieving the framework and the user of any management issues.

`RGBAPalette` class provides mapping of signal values to colors from a given palette. GUI is provided for manipulation of minimum / maximum values to be displayed and different display options are available for display of under- and over-flow bins. The palette can be imported from ROOT or specified manually.

`FrameBox` class can be used to render frames of specified dimensions and color around a set of modules of the same type. 2D and 3D frames with wire-frame or solid rendering are supported.

4.1.2. Raw-data presentation classes All raw-data presentation classes are in fact containers for individual electronic-channel representations and usually one object is used to represent one detector module. A transformation matrix (class `ZTans`) can be assigned to each object allowing the digit positions to be given in the local coordinate system. Further, by changing the position of a set of modules, they can be arranged in arbitrary layouts, not necessarily following their realistic placement in the detector. Pointers to `RGBAPalette` and `FrameBox` classes are used frequently.

`QuadSet` is the most widely used class for raw-data visualization in ALICE (used by, among others, ITS, TRD and TOF). It contains a set of rectangles, lines or hexagons (see Fig.3). For memory and rendering-speed optimization reasons, a user can specify the type of elements in a very precise way that allows almost any parameter to be held constant for the whole collection (e.g. z-coordinate, rectangle width and height, etc). For each element a signal value can be provided and is automatically mapped into a color via an `RGBAPalette` object. Additionally, an external object reference (a `TRef`) can be provided for each element.

[4] Support for arbitrary magnetic field is in preparation.

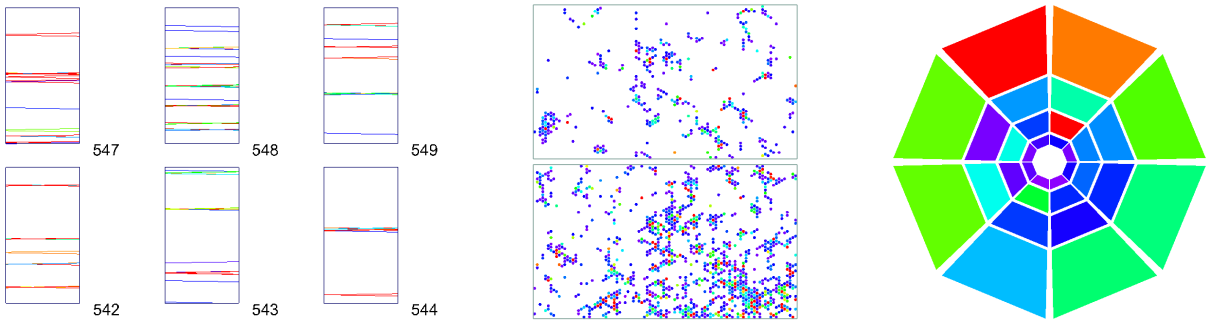

Figure 3. Examples of ALICE raw-data display using `QuadSet` class: silicon-strip detector (left), photon-multiplicity detector (middle) and VZERO (right).

`BoxSet` provides a similar service but the basic elements are 3-dimensional box-like objects defined by 8 vertices so that the variation in box-size further supplements the signal–color information. This class can be used for display of calorimeter towers and is used internally by the 3D TPC visualization.

4.2. Time projection chamber

75% of ALICE raw-data volume is taken by the TPC data, amounting to 60 MBytes for a central lead–lead event. The TPC is segmented in 36 sectors, each consisting of about 8,000 pads with 450 10-bit time-channels. The requirements for the TPC visualization include the ability to display statistical information for all pads of a given sector within the specified time-range as well as to present the pad data by expanding the time coordinate into the third dimension. Further, each pad (for 2D) and each digit (3D) needs to be selectable individually.

A sector was chosen as the basic visualization unit. To be able to respond to time-range and threshold changes in real-time a special data-storage class `TPCSectorData` had to be developed.

`TPCSectorViz` is the base class for concrete 2D and 3D sector-visualization classes. It contains controls over threshold / saturation settings for signal-to-color mapping and over the displayed time-range.

`TPCSector2D` controls the display of statistical signal values projected to the read-out plane. In addition to the base-class controls, it allows selection of the pad-data calculation algorithm which is currently limited to maximum and average value of signals in the specified time-range. Further, it controls the feedback provided upon user's selection of an individual pad. This can be a detailed printout or display of a 1D (signal vs time) or 2D (signal vs pad vs time) histogram.

`TPCSector3D` class provides a complete view and represents each digit by a box or a point in 3D space. Users can select the signal-range fraction at which the digit is rendered as a point, thus allowing them to peek inside of the digit clusters produced by a track (typical pad-row cluster sizes are 3 time-bins $\times$ 3 pads).

All classes are equipped with GUI editors. `TPCSector2D` and `3D` have custom OpenGL rendering classes supporting two-level selection. Examples are shown in Fig.4.

4.3. Inner silicon-tracker and intermediate-level raw-data inspection

During the early detector operation it is desirable to have an efficient way of browsing through sub-detector systems and make a fast survey of the recorded raw-data. This can reveal general shortcomings in the operation of detectors, read-out electronics or the DAQ event-building system and complements the high-level inspection provided by the statistical monitoring information. However, when displaying digits of fine-grained, highly-modular detectors, such as silicon trackers, one encounters two common problems:

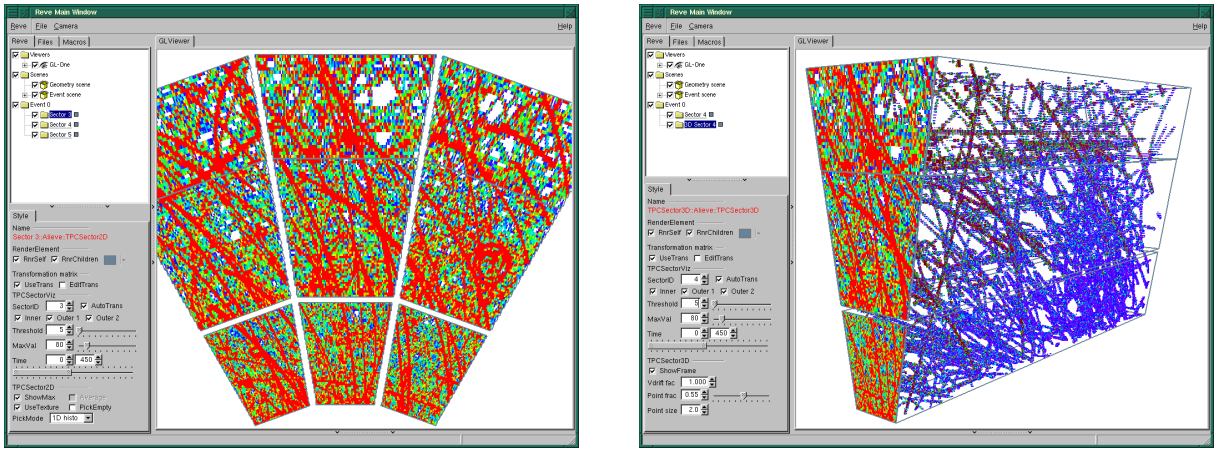

Figure 4. GUI controls and OpenGL display of TPC raw-data: `TPCSector2D` (left) and `TPCSector3D` (right).

- when modules are placed on their real 3D positions, they overlap each other and prevent simultaneous inspection of different layers;
- individual digits are too small to be visible, even at high magnification.

To overcome these issues two techniques have been employed in the `ITSModuleStepper` class that is used for intermediate-level inspection of inner-detector digits.

(i) **2D multi-paged arrangement** allows the user to simultaneously view as many modules as allowed by the screen-size and required viewing precision. They are put in a matrix layout with number of rows and columns being configurable at run-time. This naturally organizes the module collection in pages that can be traversed in random order. During initialization, a selection can be made based on module type (pixel, drift or strip detectors) and location (layer-id or ranges of φ and η).

(ii) **Digit scaling** solves the problem of small size of individual digits by merging neighbouring digits into a single visual element. The signal-value assigned to this representant is a statistical quantity calculated for the whole group, such as the average, root-mean-square, occupancy, minimum or maximum value. Several group sizes are supported (e.g. 2×2, 8×4, 32×8, etc) depending on, and varying with, the module-type segmentation.

To further streamline the inspection procedure all controls can be integrated into the OpenGL display as overlay elements, thus grouping all user-controls within a single interaction window. The comparison of *before* and *after* user-interface are shown in Fig.5.

5. Visual reconstruction validation

During the evolution of the experiment software it is important to constantly monitor its performance and reproducibility of physics results compared to older versions. Statistical tools are usually used for the high-level validation where significant shortcomings can be spotted and fixed. But with the approach of the first physics-runs it becomes increasingly important to shift the focus onto the event-level and to account for any discrepancy that might be found, down to the level of individual tracks and clusters.

To facilitate this task, a tool for visual scanning of events has been incorporated into AliEVE. In default configuration it provides a detailed view of standard reconstructed objects: primary vertex, tracks and clusters. When dealing with simulated events full backward-navigation to

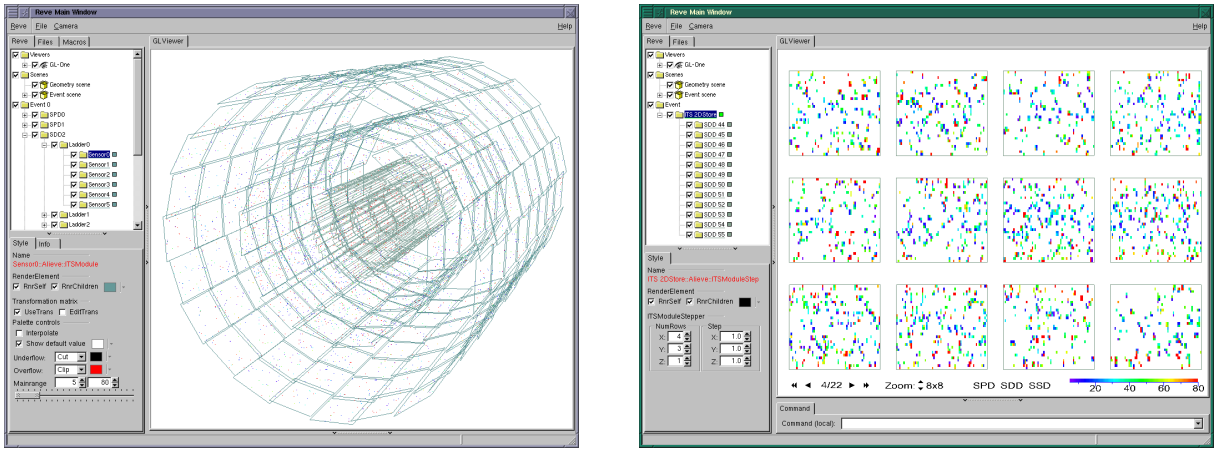

Figure 5. Comparison of ITS raw-data rendered: a) with realistic module positioning and digit-scale (left); and b) by using the `ITSModuleStepper` with 2D-page arrangement and digit scaling (right).

the simulation data is available. It can be imported as a whole or, alternatively, activated on a per-track basis.

To allow a more detailed view of the vertex region and better inspection of cluster-track associations, 2D projections and fish-eye transformations have been implemented in REVE (following the work of H. Drevermann for the ALEPH event-display [8]). The basic visualization classes have been extended to support arbitrary projections, the most relevant being: a) `GeoShapeRnrEl` – simplified geometry; b) `PointSet` – clusters and hits; and c) `Track` – simulated & reconstructed tracks and also compound objects like kinks and V0's.

The implementation of the tool is simple and easily extendible. It consists of a single dedicated class `TrackCounter`, providing graphical user-interface for common operations and management of track collections, and two macro functions. The first one is an initialization script creating the necessary objects (track-counter and projector) and setting up the event navigation. The second one is executed for each new event and loads the relevant event-data into the visualization objects. By modifying these two functions the tool can be configured and extended to suit more specific requirements. Examples of the running application are shown in Fig.6.

The visual scanning procedure has been carried out on regular intervals (every 3 months) since September 2006. Several issues with primary vertex reconstruction, track finding and also simulation itself have been identified and corrected. During the first physics runs this tool will also be used for manual track selection on an initial p–p data-sample to cross-check primary-track multiplicity, p_T and η distributions against those produced by the standard reconstruction.

6. Conclusion

ALICE event visualization environment uses a modular application core with visualization and GUI base-classes that are independent of the experiment software framework. Only specific issues are addressed in the AliROOT-dependent part of code. CINT scripts are used extensively, both for framework control and for initialization of visualization objects. By modifying, extending or combining these scripts users gain a large degree of flexibility that would be difficult to provide otherwise.

During the last year, significant development efforts have been focused on providing adequate support for detector commissioning and early detector operation. Several common solutions have been identified and implemented for the display of raw-data. Using this infrastructure, raw-

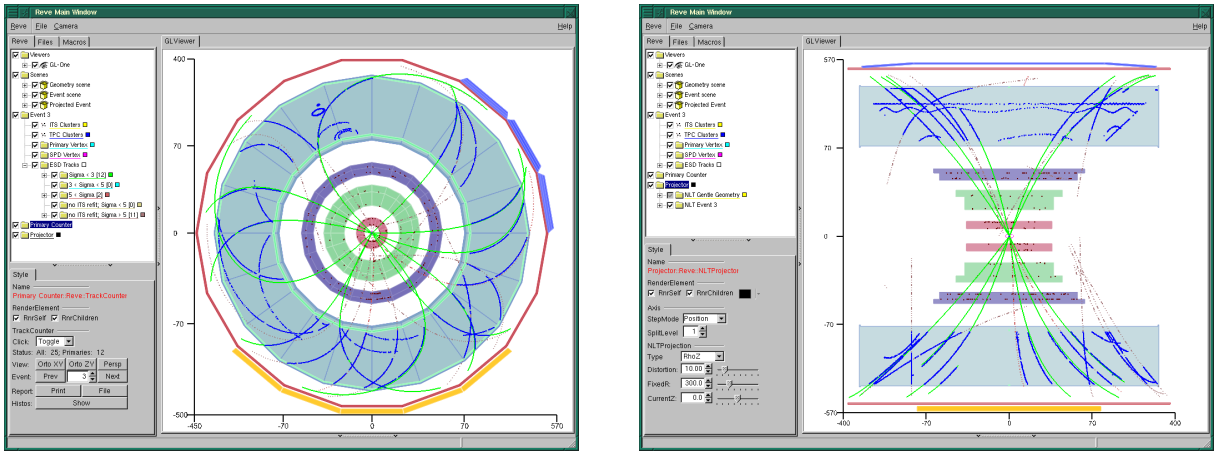

Figure 6. Examples of ALICE event-scanning interface for a p–p@900 GeV event. Left: radial fish-eye projection with `TrackCounter` GUI. Right: ϱ-z fish-eye with projection-control GUI. Central red, green and blue layers are the envelopes of the inner-tracker sub-detectors.

data visualization has been implemented for all major sub-detectors of ALICE. Complementary, framework-elements and tools allowing detailed inspection of reconstructed events have been developed to aid during the detailed validation of the full offline software chain.

REVE, the experiment-independent part of the visualization framework, is planned to be included in the ROOT distribution before its next production release, v5-18/00. It is hoped that other experiments will recognize the benefits of the common visualization infrastructure, try to use it in their environments and provide constructive feedback.

Acknowledgments

The author would like to thank Alja Mrak–Tadel for the painstaking work on the project as well as for her understanding and support, to Bertrand Bellenot for help with ROOT GUI and to René Brun for constantly reminding me about the users' needs and expectations.

References

[1] See ALICE offline project web-page: `http://aliceinfo.cern.ch/Offline/`.

[2] Brun R and Rademakers F 1997 *Nucl. Inst. & Meth. in Phys. Res.* A **389** pp 81-86. See also `http://root.cern.ch/`.

[3] Tadel M 2005 GLED – *an Implementation of a Hierarchic Server–Client Model*, (Advances in Computation: Theory and Practice vol 16) ed Pan Y and Yang L (New York: Nova Science Publishers) pp 21–37. See also `http://www.gled.org/`

[4] Tadel M and Mrak-Tadel A 2007 *XV Int. Conf. on Comp. in High Energy and Nucl. Phys. 2006* **1** (Mumbai: Macmillan) pp 398–401.

[5] Tadel M 2007 *The New Generation of OpenGL Support in ROOT* (these proceedings).

[6] Antcheva I, Brun R, Hof C and Rademakers F 2006 *Nucl. Inst. & Meth. in Phys. Res.* 1 **559** pp 17–21.

[7] Brun R, Gheata A and Gheata M 2003 A geometrical modeller for HEP *XIII Int. Conf. on Comp. in High Energy and Nucl. Phys. 2003 THMT001* arXiv:physics/0306151. See also *Root Users Guide* pp 299–350.

[8] Drevermann H, Kuhn D and Nilsson B 1995 *Event Display: Can We See What We Want to See?* Presented at CERN School of Computing '95, Arles, France. `http://ipt.web.cern.ch/IPT/Papers/CSC95/EDisplay/`