EVE - Event Visualization Environment of the ROOT framework

Matevž Tadel[*] **CERN**

E-mail: matevz.tadel@cern.ch

EVE is a high-level environment using ROOT's data-processing, GUI and OpenGL interfaces. It can serve as a framework for object management offering hierarchical data organization, object interaction and visualization via GUI and OpenGL representations and automatic creation of 2D projected views. On the other hand, it can serve as a toolkit satisfying most HEP requirements, allowing visualization of geometry, simulated and reconstructed data such as hits, clusters, tracks and calorimeter information. Special classes are available for visualization of raw-data. EVE is used in the ALICE experiment as the standard visualization tool, AliEVE, using the full feature set of the environment. In the CMS experiment, EVE is used as the underlying toolkit of the cmsShow physics-analysis oriented event-display. Both AliEVE and cmsShow are also used for the online data-quality monitoring.

XII Advanced Computing and Analysis Techniques in Physics Research
November 3-7, 2008
Erice, Italy

[*]Speaker.

1. Introduction

Visualization of detector geometry and event data plays an important role over a large spectrum of activities of a HEP collaboration, including early planning of experiments, debugging of simulation and reconstruction code, detector calibration, physics analysis and online monitoring. The extent of visualization use-cases implies that the visualization framework of an experiment must be able to handle all kinds of experiment data and present them together in a unified environment. This requirement is the main problem of visualization infrastructure, not encountered by other segments of the offline code that typically only operate on one type of input data and produce another one as their output.

Further complication of the matter arises due to the fact that visualization is expected to be both completely separate from the general offline framework as well as to be able to provide full backward navigation from derived physics objects, like jets or tracks, to more elementary ones, like clusters, and even further to raw-data or simulation records. The first expectation requires the experiment-data, or at least part of it, to be readable and interpretable with a subset of the full offline framework – the thinner this layer is, the better. The second expectation obviously requires significant presence of the core framework and, if interactive access to the algorithms is required, even complete integration of visualization code into the offline framework.

All these considerations lead to the conclusion that it is impossible to construct a single visualization program that would satisfy all the needs of a HEP collaboration. Nevertheless, there is a significant overlap of visualization elements and functionality that is common to all visualization tools, not only within one collaboration, but also within the HEP community as a whole. These common elements are exactly the targeted scope of EVE – the event visualization environment of the ROOT framework[1].

1.1 EVE overview

EVE uses ROOT GUI and OpenGL modules for its data presentation and interaction layer. On top of those it provides a visualization object-management system that integrates deep into the OpenGL infrastructure to provide for fast updates of graphical windows with minimal overhead as well as to allow fine-grained access to displayed objects and even to object-components. The visualization classes are derived from a common base class, **`TEveElement`**, that provides support for building of overlaid object-hierarchies - each element can have children and can, in turn, be registered as a child to arbitrary number of other elements.

Much of the EVE higher-level functionality, like selection-management and 3D-scenes/viewer management, is implemented as EVE classes. This facilitates the interaction with these objects (they are visible within the same GUI as the visualization elements) and allows the described element aggregation mechanism to be used on the framework level. For example, to add an element into a 3D scene, it is enough to drag'n'drop it into the element representing the scene.

All EVE visualization classes allow backward references to the experiment data they are representing by providing a `TRef` per visualized entity. This can correspond to an EVE element (e.g. **`TEveTrack`**) or to an entry in an element if its class is actually a container (e.g. **`TEvePointSet`**, used to represent hits, clusters and other point-like data). In this way the visualization classes can be completely separated from the experiment framework and still provide full backward-navigation.

1.2 Development history

Development of EVE started in 2005 with prototyping of the visualization elements and selection algorithms within the GLED framework [2]. This led to the selection of OpenGL as the only supported drawing library. In 2006, the first prototype using the ROOT graphical user-interface and 3D graphics was assembled using AliRoot, the offline framework of the ALICE collaboration, as the development platform [3]. OpenGL support in ROOT has also been augmented as a part of this development [5].

In December 2007, EVE was introduced into ROOT as a standard module. Since then, several experiments besides ALICE have already started using it, including CMS, FAIR, NA-62 and T2K. EVE reached maturity towards the end of 2008 and no major changes in the interfaces are foreseen. Several extensions and optimizations of the current functionality are planned for 2009 and 2010.

2. EVE architecture

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

1. **Application core** consists of general services exposed to users via an instance of the **`TEveManager`** class and static functions of the **`TEveUtil`** class. Together, they provide the following main functionalities:
   - management of object-browsers, 3D scenes and 3D viewers;
   - management of ROOT GUI windows in arbitrary configurations of tabs, stacks and main-frames;
   - registration of visualization objects;
   - selection management;
   - data-base of visualization parameters that can be assigned to elements based on their type (a string tag);
   - event management & navigation;[1]
   - execution environment for CINT scripts.
2. **Framework base-classes** implement the low-level functionality of visualization and GUI objects that are 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 name, title and color, object highlighting and selection, object inspection via GUI and command-line interfaces).
3. **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, tracks and calorimetry) are discussed in Sec.3. Most of these classes can be automatically projected into 2D views ($r-\varphi$ and $\rho-z$), see Sec.5. Classes for raw-data visualization are described in Sec.4.

[1]Due to the variability of input-data, only a very basic infrastructure can be provided for this task.

All classes in this category are equipped with accompanying GUI editors and OpenGL rendering classes. Following ROOT's naming convention, the **`TEveTrack`** class has the accompanying GUI implemented in class **`TEveTrackEditor`** and the GL drawing functions in class **`TEveTrackGL`**. For details of this mechanisms see [5] and [6].

3. Standard visualization classes

In this category we describe the traditional event-visualization classes of EVE that must be present, in one form or another, in 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.

3.1 TEveElement – the visualization base-class

`TEveElement` is the base-class of all visualization classes in EVE. It provides the interfaces between the objects and the application core, the 3D rendering system and the 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]. EVE supports two methods for geometry visualization: the first one uses TGeo directly and the second one presents pre-extracted volume-shape tessellations.

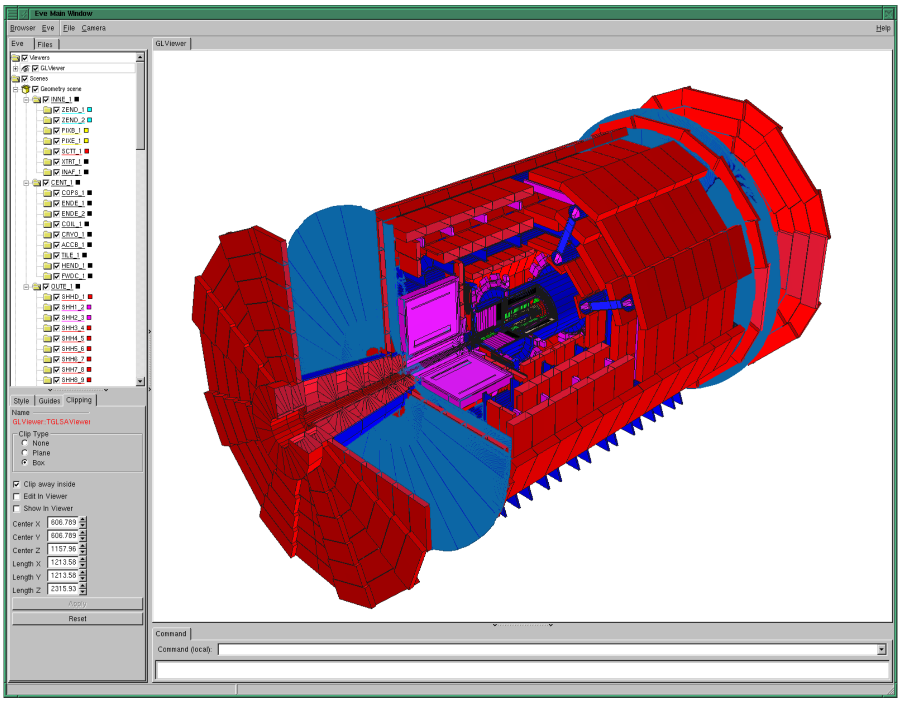
3.2.1 Display of full TGeo geometries

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

The **`TEveGeoNode`** class allows further link between TGeo and EVE by providing representation of children nodes in the EVE 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. Example of such geometry is shown in Fig.1.

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 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



Name GLViewer::TGLSAViewer

Clip Type: ( ) None ( ) Plane [✓ Box]

[✓] Clip away inside
[ ] Edit In Viewer
[ ] Show In Viewer

Center X	606.789
Center Y	606.789
Center Z	1157.96
Length X	1213.58
Length Y	1213.58
Length Z	2315.93

Figure 1: Geometry of the ATLAS detector in EVE. The object browser shows the node hierarchy. Clipping box is used to provide view into the central region of the detector.

`TEveGeoShape` 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. For examples, see Fig.2 and Fig.3.

3.3 Hits & clusters

Hits and clusters can be visualized by using the **`TEvePointSet`** 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 **`TEvePointSet`** can own the reference objects and delete them upon its destruction).

To fill the data, a sequential method can be used, specifying coordinates and an external reference for each point in turn, or use a special **`TEvePointSelector`** class that invokes the full ROOT machinery for selecting data from `TTree`'s.

The **`TEvePointSetArray`** class implements an interactive 3D-histogram by encompassing an array of **`TEvePointSet`**'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.

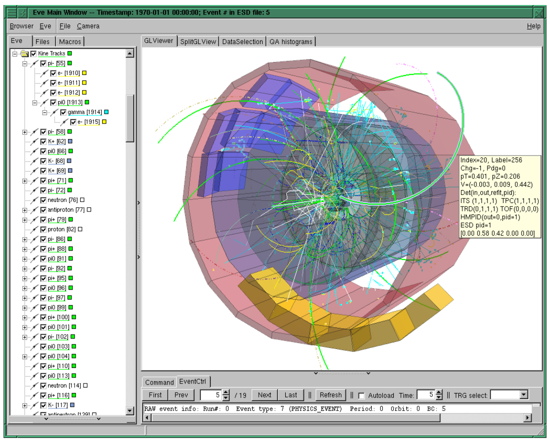
Several examples can be seen in Fig.2 and Fig.3.

3.4 Trajectories, particles & tracks

The **`TEveTrack`** class can be used to represent particle trajectories. An arbitrary number of control-points can be specified along the track, to mark one of: a) position/momentum reference, b) daughter creation point, c) decay point, and d) reconstructed cluster. Extrapolation and interpolation in an arbitrary, user-provided, magnetic field is supported by the service class **`TEveTrackPropagator`**, which also contains general track visualization parameters (e.g. maximum radius and z-coordinate of extrapolation, required precision, etc). Usually, all tracks from a given data-source reference the same track-propagator 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 **`TEveTrackList`** object that provides control over common track rendering parameters and interactive selection of displayed momentum ranges.

Several examples are shown in Fig.2 and Fig.3. Note also the object browsers on the left side of the shown windows and track highlighting that can be used to display track-parameters and other relevant information in a tool-tip.


Eve Main Window -- Timestamp: 1970-01-01 00:00:00; Event # in ESD file: 5
GLViewer
SplitGLView
DataSelection
QA histograms

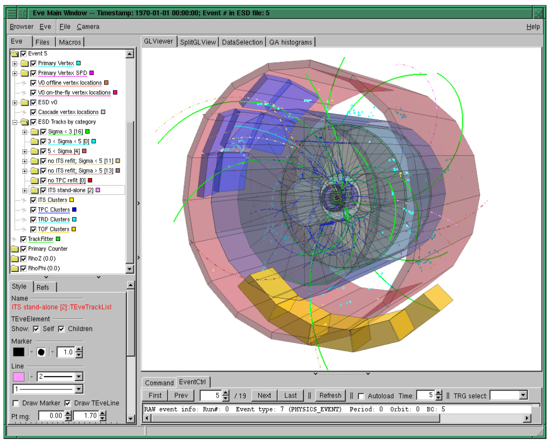
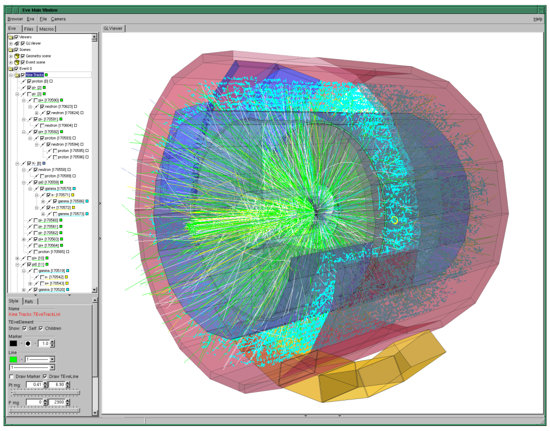

Eve Main Window -- Timestamp: 1970-01-01 00:00:00; Event # in ESD file: 5
GLViewer
SplitGLView
DataSelection
QA histograms

Figure 2: An ALICE p–p@14 TeV event showing simulated (left) and reconstructed data (right).



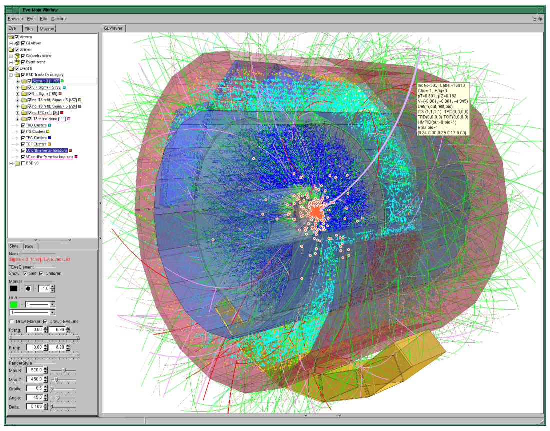


Figure 3: An ALICE Pb–Pb@5.4 TeV/nucleon event showing simulated (left) and reconstructed data (right).

3.5 Calorimeter data

Calorimeter classes in EVE can be divided into two categories: the data holding classes and the visualization classes.

3.5.1 Data holding classes

The calorimeter visualization must support advanced operations on the calorimeter data, like automatic rebinning and summing of calorimeter cells, in η and φ, as well as along the longitudinal segmentation. Further, it must be possible to set separate thresholds on each longitudinal segment of the calorimeter. All these operations are available via the abstract interface **`TEveCaloData`**. Two concrete implementations are provided in EVE.

`TEveCaloDataHist` represents the data in a stack of energy versus η-φ histograms, internally using the ROOT's `THStack` class. Each histogram in the stack represents one longitudinal segment. This is convenient for global views of the calorimeter response as well as for physics analysis applications.

`TEveCaloDataVec` allows the user to specify individual calorimeter cells by their η and φ range and longitudinal segment identification. This class can be used for detailed views as there is no assumption on the η-φ segmentation of the individual cells.

3.5.2 Visualization classes

The visualization classes reference the data-holding object and present a specific representation of this data. Several visualization objects can point to the same data-object and thus provide a coherent view, based on the thresholds and other visualization parameters specified by the user.

The abstract base-class **`TEveCaloViz`** provides the data-members and methods that are common to all visualization classes: η and φ ranges, scales (absolute or relative), a flag whether E or E_T should be displayed, and optionally a palette for mapping of signal value to color.

`TEveCalo3D` displays the calorimeter in 3D space, as a set of towers. An example is shown on the left side of Fig.4.

`TEveCaloLego` displays the η-φ lego-plot, which is the preferred way to view calorimeter data for physics analysis. The class supports two rendering modes: as a 3D histogram (right side of Fig.4) and as a 2D plot, where sizes or colors of the cells represent the deposited energy (bottom-left window on Fig.11).

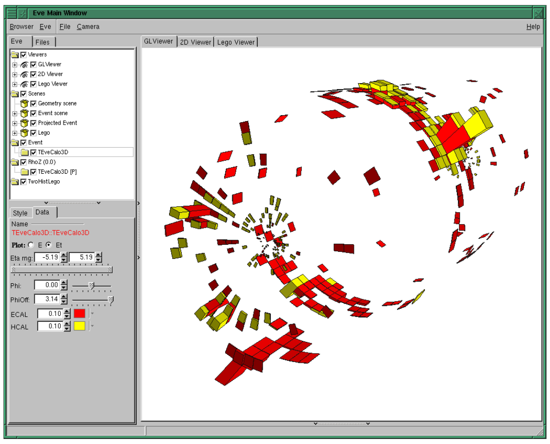
`TEveCalo2D` is used to present the cells summed up in η or φ, and is used in projected views (right picture in Fig.9).

3.6 Miscellaneous visualization classes

`TEveText` allows arbitrary text to be displayed in 2D or 3D. The FTGL library is used internally to display the text. Extruded 3D fonts are also suported.

`TEveTriangleSet` displays triangle meshes and each triangle can be assigned its own color. A complex example is shown in Fig.5.

`TEveArrow` provides display of a 3D arrow. This can be used to represent directions in space (like missing E_T) or to point to a specific feature in 3D space.



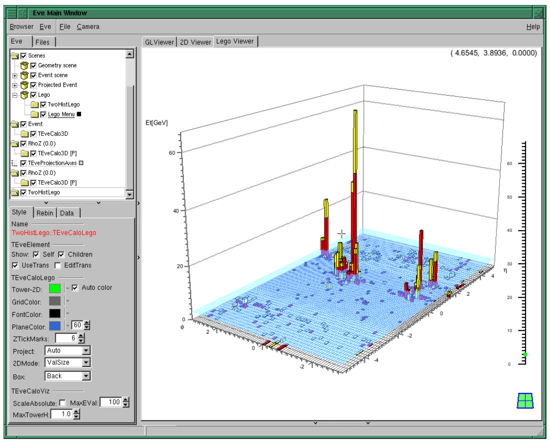


Figure 4: Calorimeter classes of EVE: **`TEveCalo3D`** (left) and **`TEveCaloLego`** (right).

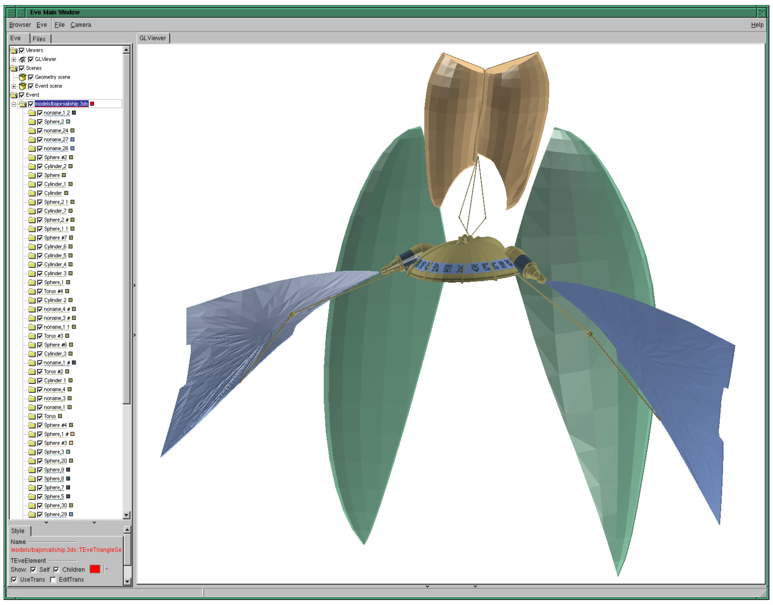


Figure 5: **`TEveTriangleSet`** showing a 3D-studio model (`3ds` importer provided by B. Bellenot).

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 support classes for raw-data visualization are discussed first and then classes for raw-data presentation are described in detail.

4.1 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 objects are reference-counted with automatic destruction, thus relieving the framework and the user of any management issues.

The **TEveRGBAPalette** class provides mapping of signal values to colors from a given palette. A GUI is provided to manipulate 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.

The **TEveFrameBox** 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.

4.2 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 **TEveTrans**) can be assigned to each object allowing the position of the digit 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. All this basic functionality is provided in the base-class **TEveDigitSet**, including the pointers to **TEveRGBAPalette** and **TEveFrameBox** objects. For each digit, a signal value and an external object reference (a `TRef`) can be specified by the user. A histogram of registered signals can be produced via a GUI button.

TEveQuadSet is the most widely used class for raw-data visualization. It contains a set of rectangles, lines or hexagons (see left side of Fig.6). 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).

TEveBoxSet provides a similar service but the basic elements are 3-dimensional box-like objects or cones so that the variation in size further supplements the signal–color information. See the right side of Fig.6.

For more complex examples custom classes need to be designed. For example, ALICE TPC is expected to produce 80 MBytes of raw-data per central Pb-Pb collision and optimised algorithms are required to control visualization parameters. Screen-shots of TPC visualization are shown in Fig.7.

5. 2D projections

2D projections and fish-eye transformations are indispensable for a detailed inspection of the vertex region as well as for good utilization of the screen-estate as, without some artificial scaling, outer detectors typically use up much more space than inner ones [8]. In EVE, the projections can be performed automatically on extracted geometry, points, tracks and on calorimeter data. Currently EVE provides r-φ and ρ-z projections, but the interface is not restricted and supports easy addition of new projection-types.

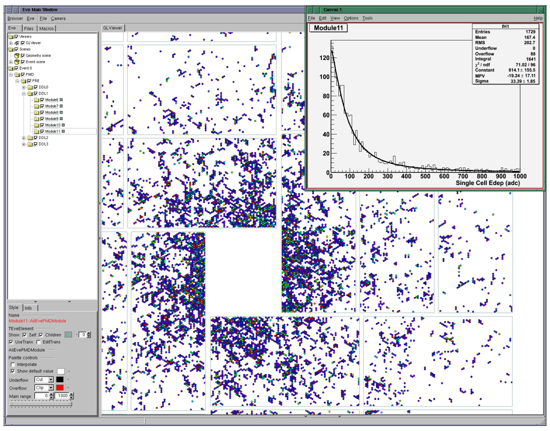
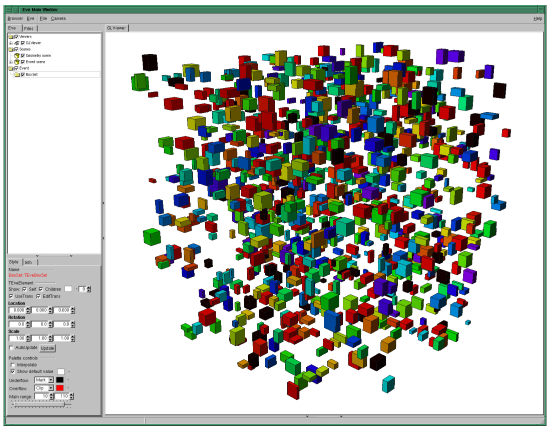



Figure 6: Examples of raw-data visualization classes. Left: **`TEveQuadSet`** as used for ALICE PMD detector. Right: **`TEveBoxSet`** filled with random data.

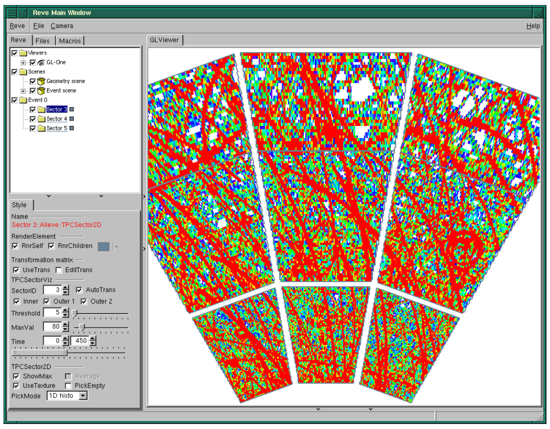
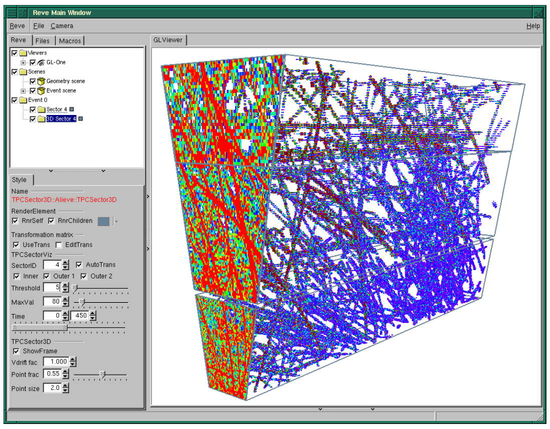



Figure 7: Visualization of ALICE TPC data for a simulated Pb–Pb event. Separate sub-classes were developed for display of the data in 2D (left and right) and 3D (right).

The central management of a given projection is done by the **`TEveProjectionManager`** class. It connects together the actual geometric projection (abstract base-class **`TEveProjection`**) and a list of EVE-elements that need to be projected.

EVE elements that support the projection operation must derive from the abstract interface **`TEveProjectable`**. One of the main roles of this class is to specify the class of the projected object (abstract method `TClass* ProjectedClass()`). This is used by the manager to instantiate a replica of proper type. Further, the projectable also holds the list of projected replicas to allow for propagation of changes in visualization parameters.

Another abstract interface is required for the projected classes – **`TEveProjected`**. This allows the manager to, first, inform the projected object about its original (projectable) object and, second, to update it when the projection parameters change.

Examples of projected views are shown in figures 8 and 9.

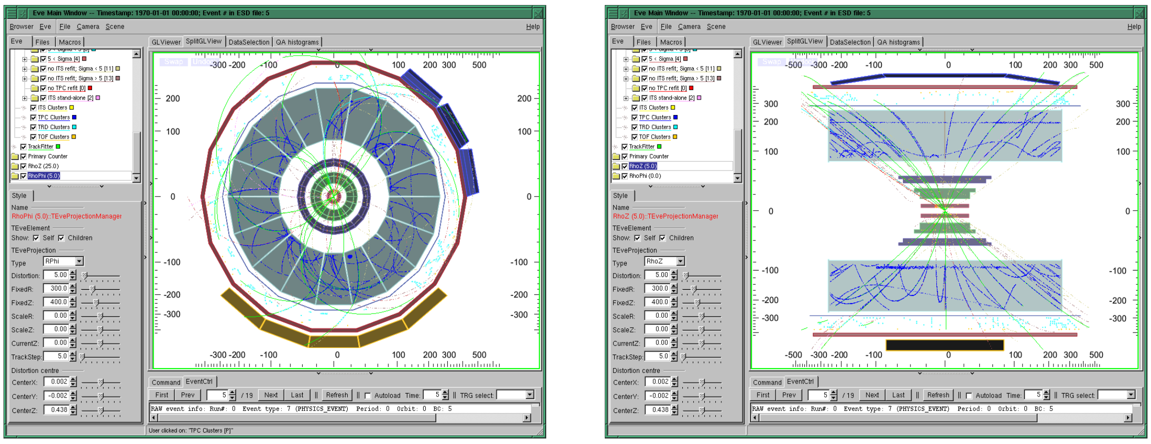


Figure 8: r-φ (left) and ρ-z (right) projections of a ALICE pp event. Fish-eye distortion is enabled in both views.

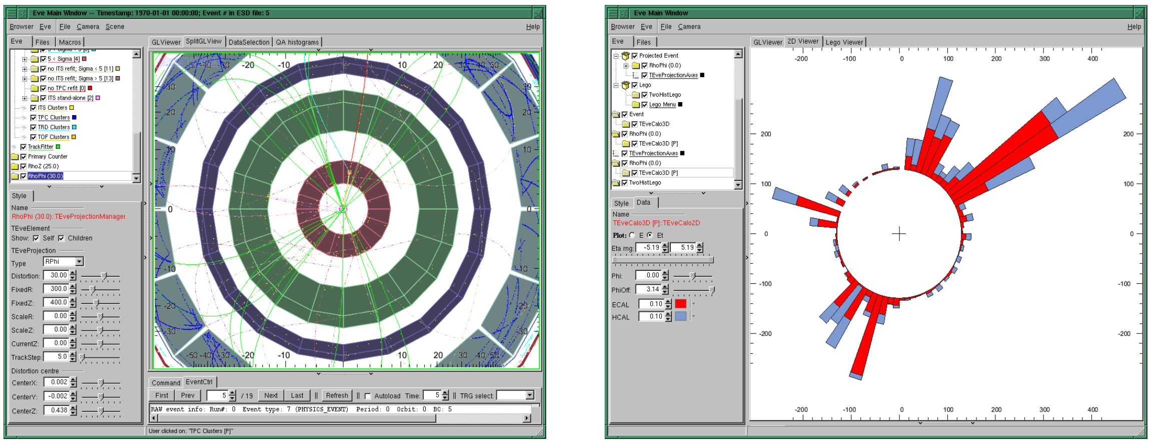


Figure 9: Left: close-up on the silicon tracker of the event shown in the previous figure. Right: r-φ view of a CMS event using the **`TEveCalo2D`** class. Red and light-blue segments represent the EM and hadronic calorimeters.

6. Usage of EVE

6.1 AliEVE – ALICE Event Visualization Environment

AliEVE is the standard visualization tool of the ALICE experiment. It is used not only for standard offline visualization tasks, but also for physics analysis, high-level trigger visualization and for online monitoring. Its functionality can be split into three main elements.

1. **Access to ALICE data and core classes.** 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 field maps, alignment data and detector-conditions database.

This functionality is aggregated into the **EveBase** module. It depends on a minimal set of AliRoot's components and is required for visualization of ESD and AOD data.

2. **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 the TPC, in part also due to its large data-volume. Other complex detectors (e.g. ITS, TRD, TOF) extend EVE 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 EVE classes directly.

3. **Visualization scripts** are CINT macros that perform the actual extraction of the data, 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.

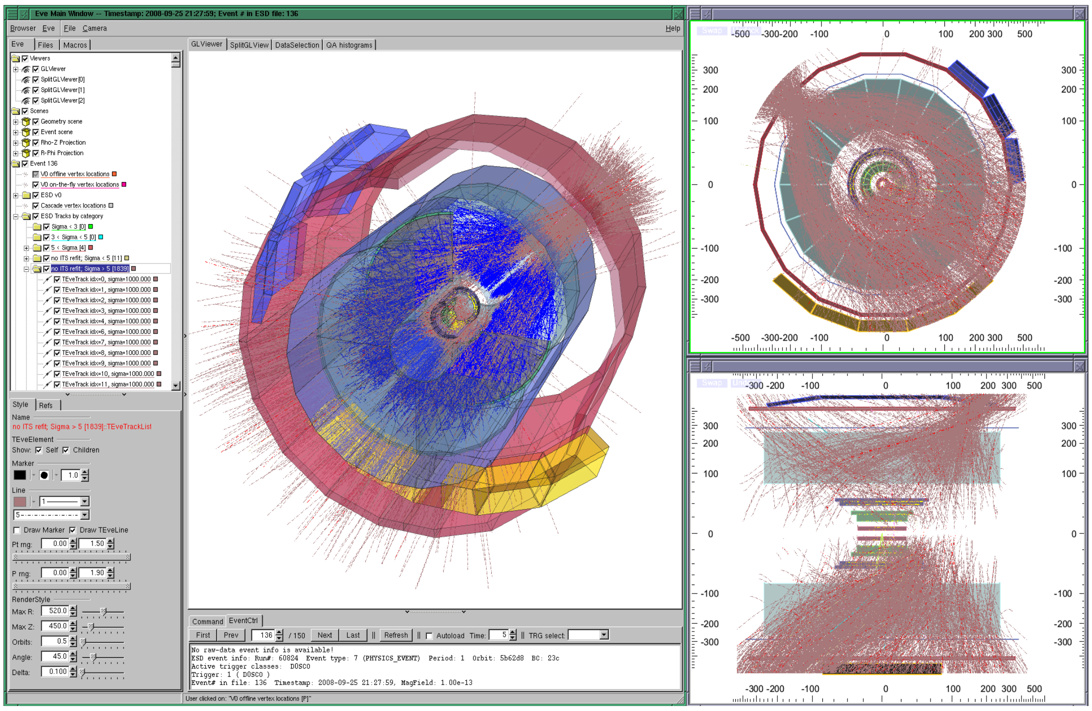
Every effort is made to keep AliEVE as small and as simple as possible. Another simplification comes from the usage of CINT scripts for data-extraction steering. By providing a concise interface for their invocation and exception-throwing methods for obtaining handles into the ALICE data, their framework 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 to understand the macros, and to tailor or enhance them for their specific needs without any further complications. A screen-shot of AliEVE in action is presented in Fig.10. More details about AliEVE are available in [4].

6.2 cmsShow – Physics Analysis Oriented Event Display of CMS

cmsShow is based on the light version of the CMS software framework and uses EVE as a toolkit to present and manage its visualization data. Most of the EVE GUI functionality is hidden from the user to prevent users from wandering too deep into the structure of EVE. All supported operations are provided via a concise and consistent user interface that makes it easier to use for novice and non-expert users. However, EVE components are used extensively and several high-level features of EVE were designed in collaboration with cmsShow developers. The most prominent examples are the calorimetry visualization classes and the selection-feedback system. An example of cmsShow is shown in Fig.11.

7. Conclusion

EVE is a complete visualization framework, satisfying a full range of requirements of a HEP collaboration. It is distributed within ROOT, which is already used by most HEP collaborations, at least for data-processing and data-analysis. Thus the usage of EVE becomes even more convenient


Eve Main Window -- Timestamp: 2008-09-25 21:27:59; Event # in ESD file: 136
Browser Eve File Camera Help
Eve Files Macros
GLViewer SplitGLView DataSelection QA histograms
Command EventCtrl
First Prev 136 / 150 Next Last Refresh Autoload Time: 5 TRG select:
User clicked on: "V0 offline vertex locations [P]"

```
No raw-data event info is available!
ESD event info: Run#: 60824  Event type: 7 (PHYSICS_EVENT)  Period: 1  Orbit: 5b62d8  BC: 23c
Active trigger classes:  D0SC0
Trigger: 1 ( D0SC0 )
Event# in file: 136  Timestamp: 2008-09-25 21:27:59, MagField: 1.00e-13
```

Figure 10: A composite screenshot of AliEVE showing a cosmic shower observed during the cosmic-ray runs in summer 2008.

as it does not encumber the visualization environment with additional software requirements that might hinder deployment and portability.

Wide and fast adoption of EVE in the HEP community shows that it has indeed provided a missing piece in the landscape of HEP visualization software. Future work on EVE will mostly focus on implementation of features and extensions required by the user community.

Acknowledgements

The author would like to thank Alja Mrak–Tadel for the dedicated work on the project as well as for her understanding and support.

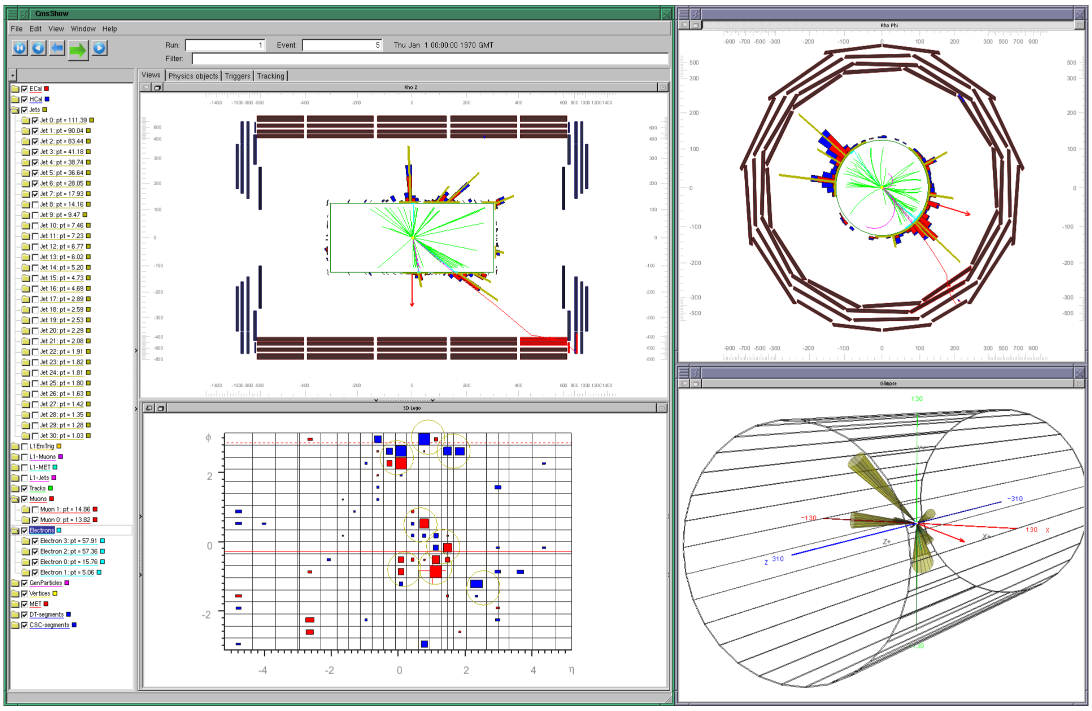
References

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

[2] M. Tadel 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/`

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

[4] M. Tadel, *Raw-data display and visual reconstruction validation in ALICE*, J.Phys.Conf.Ser.119:032036,2008.



Run: 1 Event: 5 Thu Jan 1 00:00:00 1970 GMT
Filter:

- [✓] ECal
- [✓] HCal
- [✓] Jets
  - [✓] Jet 0: pt = 111.39
  - [✓] Jet 1: pt = 90.04
  - [✓] Jet 2: pt = 83.44
  - [✓] Jet 3: pt = 41.18
  - [✓] Jet 4: pt = 38.74
  - [✓] Jet 5: pt = 36.64
  - [✓] Jet 6: pt = 28.05
  - [✓] Jet 7: pt = 17.93
  - [ ] Jet 8: pt = 14.16
  - [ ] Jet 9: pt = 9.47
  - [ ] Jet 10: pt = 7.46
  - [ ] Jet 11: pt = 7.23
  - [ ] Jet 12: pt = 6.77
  - [ ] Jet 13: pt = 6.02
  - [ ] Jet 14: pt = 5.20
  - [ ] Jet 15: pt = 4.73
  - [ ] Jet 16: pt = 4.69
  - [ ] Jet 17: pt = 2.89
  - [ ] Jet 18: pt = 2.59
  - [ ] Jet 19: pt = 2.53
  - [ ] Jet 20: pt = 2.29
  - [ ] Jet 21: pt = 2.08
  - [ ] Jet 22: pt = 1.91
  - [ ] Jet 23: pt = 1.82
  - [ ] Jet 24: pt = 1.81
  - [ ] Jet 25: pt = 1.80
  - [ ] Jet 26: pt = 1.63
  - [ ] Jet 27: pt = 1.42
  - [ ] Jet 28: pt = 1.35
  - [ ] Jet 29: pt = 1.28
  - [ ] Jet 30: pt = 1.03
- [ ] L1EmTrig
- [ ] L1-Muons
- [ ] L1-MET
- [ ] L1-Jets
- [✓] Tracks
- [✓] Muons
  - [ ] Muon 1: pt = 14.86
  - [✓] Muon 0: pt = 13.82
- [✓] Electrons
  - [✓] Electron 3: pt = 57.91
  - [✓] Electron 2: pt = 57.36
  - [✓] Electron 0: pt = 15.76
  - [✓] Electron 1: pt = 5.06
- [✓] GenParticles
- [✓] Vertices
- [✓] MET
- [✓] DT-segments
- [✓] CSC-segments

Figure 11: A screenshot of cmsShow displaying a simulated $t\bar{t}$ event.

http://dx.doi.org/10.1088/1742-6596/119/3/032036

[5] M. Tadel, 2007 *The New Generation of OpenGL Support in ROOT*, J.Phys.Conf.Ser.119:042028,2008. http://dx.doi.org/10.1088/1742-6596/119/4/042028

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

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

[8] H. Drevermann, D. Kuhn and B. Nilsson, *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/`