The New Generation of OpenGL Support in ROOT

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

Abstract.
OpenGL has been promoted to become the main 3D rendering engine of the ROOT framework. This required a major re-modularization of OpenGL support on all levels, from basic window-system specific interface to medium-level object-representation and top-level scene management. This new architecture allows seamless integration of external scene-graph libraries into the ROOT OpenGL viewer as well as inclusion of ROOT 3D scenes into external GUI and OpenGL-based 3D-rendering frameworks.

Scene representation was removed from inside of the viewer, allowing scene-data to be shared among several viewers and providing for a natural implementation of multi-view canvas layouts. The object-graph traversal infrastructure allows free mixing of 3D and 2D-pad graphics and makes implementation of ROOT canvas in pure OpenGL possible. Scene-elements representing ROOT objects trigger automatic instantiation of user-provided rendering-objects based on the dictionary information and class-naming convention. Additionally, a finer, per-object control over scene-updates is available to the user, allowing overhead-free maintenance of dynamic 3D scenes and creation of complex real-time animations. User-input handling was modularized as well, making it easy to support application-specific scene navigation, selection handling and tool management.

1. Introduction

ROOT, the object-oriented data-analysis framework[1], has been growing in popularity since its introduction in 1996 and is today a de-facto standard tool for data-analysis and data-presentation within the *high-energy physics* (HEP) community. Together with the growth of its user community, ROOT also grew in scope and versatility[2]. Today it features, from a viewpoint of this article, a complete operating-system interface (including multi-threading and networking), an internal multi-platform graphical user-interface toolkit and an actively developed system for 3D visualization.

OpenGL is a versatile API for developing portable, interactive 2D and 3D graphics applications. It has an open specification that allows large parts of its primitive-processing pipeline to be implemented in graphics hardware. OpenGL was introduced in 1992 and since then slowly gained in popularity as the once expensive graphics processors became more and more affordable.

The conditions for acceptance of OpenGL into the HEP community were a bit unfortunate. At the time when OpenGL was becoming widely used, all major HEP centers adopted GNU/Linux as their primary computing platform. Introduction of hardware-accelerated OpenGL support into GNU/Linux was at best sporadic and still presents a problem for standard deployment to user PCs. However, the main stumbling blocks for wide adoption of OpenGL in HEP are mostly

removed and CERN already provides vendor specific graphics-card drivers in its *Scientific Linux CERN 4* package repository. Following these developments, OpenGL is now becoming the main 3D graphics engine of the ROOT framework.

2. Previous work

The first 3D graphics implementation in ROOT was provided by N. Bunčić in 1997. It supported Geant3 shapes, markers and poly-lines as rendering primitives and two output devices: direct painting into the standard ROOT graphics-pad and external, stand-alone viewing using the X3D library.[1] In 2000 the first version of ROOT OpenGL viewer, supporting the above primitives, was written by V. Fine and then extended to incorporate an OpenInventor-compatible library. Due to portability problems, the OpenInventor support has been removed from the standard ROOT distribution in 2004, but the work continued within the STAR collaboration.

A major development of OpenGL infrastructure in ROOT was done in 2005 by R. Maunder & T. Pocheptsov. It was recognized that, unlike traditional 3D engines, OpenGL can benefit greatly by keeping an internal database of common data for replicated scene elements, especially as they can be stored on the graphics hardware. This allowed interactive rendering of complicated detector geometries, at the order of millions of geometric primitives, with interactive refresh-rates. Furthermore, as all geometric data was stored within the OpenGL engine, it was possible to manipulate camera position, orientation and field of view, as well as other viewer dependent elements such as clipping planes, independently and without further access to original objects.

All data transfer from producer objects to the viewer relied on the `TVirtualViewer3D` API and used the `TBuffer3D` structure as the data envelope.[2] The benefit of that was that producer classes specify their 3D representation, but do not care how it is being rendered. However, it was not possible to cover all the cases with the `TBuffer3D` and possible graphical representations were limited to those required for the rendering of detector geometry.

These developments introduced many impressive features into the scene rendering pipeline that further increased the rendering speed and improved the user experience:

- **view-frustum culling** reduces the number of primitives rendered during each render-pass by skipping the primitives lying outside the viewing volume;
- **view-dependent level-of-detail** allows the primitives to be rendered at a lower precision as their screen-size diminishes, thus reducing the polygon-count processed on each render-pass;
- **support for composite shapes** by using the *constructive solid geometry* to perform boolean operations on primitive shapes;
- **user-specified clipping objects** allow users to remove part of a scene on one side of a plane or inside of a box.

The detailed status of ROOT OpenGL at the beginning of 2006 is presented in [3]. All this functionality has been retained, and even augmented, during the latest developments.

3. Overview of the new developments

The above OpenGL infrastructure was used to prototype the ALICE Event Visualization Framework (AliEVE) in early 2006 [4]. The large data-content of ALICE lead–lead events poses a big challenge on the visualization framework as it is imperative to be able to select and modify collections of event-data, like tracks, hits or clusters, that match certain criteria. Additionally,

[1] X3D viewer, written by M. Spychalla in 1992. Unrelated to X3D, the ISO standard for real-time 3D computer graphics, the successor of VRML.

[2] At the same time, the `TBuffer3D` transfer mechanism was also ported to pad and X3D viewers.

to display raw-data and digits for a selection of detector modules and signal ranges, a specific graphic representation with the ability to provide both fast rendering and backward-navigation to the original data is needed.

As the existing OpenGL infrastructure in ROOT was designed mostly to support viewing of static geometry with many replicas of original geometric volumes, we encountered three main performance and flexibility issues that needed to be addressed.

(i) **Viewer–scene entanglement.** Viewer was using a single scene as a container to store all objects and the only way to interact with a scene was via the viewer itself. Additionally, scene was aware of the viewer and used its API to manage object selection and settings of the clipping object. Thus one viewer could only display a single scene and each scene was tightly bound to a given viewer.
In an event-display application one aspires to be able to combine several independent scenes (e.g. geometry-scene, track-scene, cluster-scene) in an arbitrary arrangement of independent viewers, each of them showing a different collection of scenes with different viewing parameters. To achieve this goal the whole class architecture of the viewer and scene had to be revisited and carefully restructured, including steering of the rendering process and selection management. This is described in Sec.4.

(ii) **Scene-updates drop all internal state.** As the only way to access and change scene-data was via the viewer API, it was not possible to apply modifications to scene objects or to add or remove a sub-collection of objects. Instead, the whole scene-data was dropped on each update and all data reloaded.
This introduced a large overhead for event-display applications where it is frequent to change visibility or markup of a relatively small number of selected objects. The restructuring of scene–viewer relationship allowed as to also address this issue. An API for fine-grained updates of scene-elements was added to the scene class.

(iii) **Restriction on format and structure of supported primitives.** The previous viewer architecture was using `TBuffer3D` to transfer all tessellation data from original object to its visual representation.[3] While this allowed complete decoupling of the tessellation producer from the viewer, it also limited available visual representations of an object. This became particularly restricting for visualization of raw-data. To overcome this limitation we extended the viewer-side to be able to instantiate a special rendering object, based on the dictionary information of the producer class, that performs direct OpenGL calls during the rendering of the scene. For details see Sec.5.
Additionally, there was no infrastructure for deeper inspection of the rendered objects. For example, if you had a collection of clusters it was impossible to select one of them, it was always the whole collection that got selected. Finer granularity would have been possible by instantiating one render-object per collection element but this would have lead to waste of resources as the overhead of the infrastructure is about 512 bytes per object. The solution to this problem was to extend the direct-rendering infrastructure with a virtual interface for performing a two-level selection as described in Sec.6.

To generalize the existing infrastructure for object and clipping-plane manipulation we introduced a viewer-specific list of *overlay elements* that allow direct, low-overhead user-interaction with the viewer elements and implementation of GUI widgets within the OpenGL window (see Sec.7).

[3] Some specific shapes like spheres and cylinders are supported natively by the viewer, so just the shape-parameters are actually transferred.

4. Restructuring of scene and viewer class structure

There were three main requirements guiding the restructuring of the viewer and scene classes.

(i) Each scene must be an independent entity, regardless of how many viewers are displaying it. At the same time each scene must maintain a single display-list element for each visual as well as retain the ability to cache per-viewer information of visible objects.

(ii) Scenes must support fine grained updates, like refreshing, adding or removing of a single scene-element, without dropping neither scene-state nor the data cached within the OpenGL engine.

(iii) It must be possible to include external rendering or scene-graph libraries into the ROOT rendering pipeline.

(iv) Conversely, ROOT scenes must be made as independent as possible from the rest of the viewer infrastructure to make it possible to include them into external rendering frameworks.

In the first step of the reimplementation the elementary parts of scene and viewer classes were decoupled from the original code and put into base-classes `TGLSceneBase` and `TGLViewerBase`. After that, the remaining code was cleaned-up and API for fine-grained scene updates was added. In the last step the infrastructure that supported the `TVirtualViewer3D` API was moved into a separate class `TGLScenePad`. The overall structure of the new classes is shown in Fig.1.

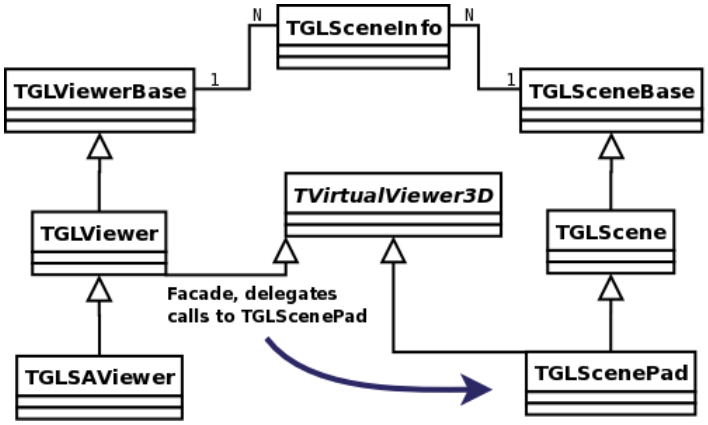

Figure 1. UML diagram of the viewer and scene classes. The `TGLSceneInfo` class allows many-to-many relationship between viewer and scene classes while retaining the specific, per-scene object caches to be optimized for particular scene-type.

4.1. Scene classes

TGLSceneBase contains virtual interfaces used by viewers, the most important ones being rendering methods and query for the scene's bounding-box allowing proper initialization of viewing-parameters and some high-level optimizations. Additionally, a list of viewers that contain the given scene is maintained so that scene changes can be propagated to the viewers for update registration.

There is no provision for element containers and no assumptions, other than the bounding-box extent, are made about the content. By sub-classing `TGLSceneBase` it is possible to plug arbitrary complex scenes, including scenes produced by external high-level rendering libraries or frameworks, into the ROOT viewer.

TGLScene is, in its essence, a container for ROOT's native representation of rendering elements, logical and physical shapes. It is a cleaned-up version of the old scene implementation and also supports fine-grained object updates.

To export a ROOT scene into a foreign rendering system, one would have to wrap this class according to the requirements of the said framework. To fully use view-dependent rendering

optimizations of the ROOT scenes, one should also provide viewing parameters via the render-context class, as described in Sec.4.4.

TGLScenePad contains the specific interface used by standard ROOT graphics classes. First, it implements the `VirtualViewer3D` interface and thus naturally services the traditional method of element registration. Second, it can take a ROOT `TPad` as an argument and autonomously iterate through its contents.[4]

4.2. TGLSceneInfo and sub-classes

The old scene implementation contained view-specific list of visible elements that, in certain conditions, improved the rendering speed. Additionally, it held a set of scene-rendering parameters, most relevant being clipping planes, required level-of-detail and rendering style (solid, wire-frame or outline). These can not be kept neither in scene, nor in viewer as they pertain to a *scene being shown in a viewer*. The `TGLSceneInfo` class provides a solution to this problem and assembles view-dependant scene parameters into a single structure.

More specific scene classes might require a more elaborate view-dependant information and thus it is left to the scene itself to create an appropriate scene-info structure by overriding a virtual function `TGLSceneInfo* TGLSceneBase::CreateSceneInfo(TGLViewerBase* view)`.

4.3. Viewer classes

TGLViewerBase is a minimal abstraction of the old viewer containing a collection of scenes, virtual interface for render-steering and camera information.

TGLViewer extends the basic viewer with interfaces for event-handling, selection management (including two-level selction) and overlay event-handling. This class is already ROOT specific as it processes standard GUI events and uses the signal-slot mechanism to notifiy the encompassing application. For backward-compatibility the `TGLViewer` class still implements the `TVirtualViewer3D` interface. The implementation is really just a facade and all calls are actually handled by the appropriate instance of the `TGLScenePad` class.

TGLSAViewer is a top-level, stand-alone viewer with concrete graphical user-interface that was already present in the old implementation. Remodularization of its structure, allowing further reuse of GUI elements, was performed.

4.4. Rendering and OpenGL context classes

TGLRnrCtx class is a covering structure allowing general render-parameters to be passed from a viewer to scenes and further down into rendering functions of visual elements. It contains references to the current viewer, scene-info and scene objects, camera information, selction mode, active clipping planes, required level of detail and rendering style. This allows specific rendering code to access all information that can be relevant for rendering, e.g. coordinate axes require camera information for proper rendering of tick-marks and labels.

TGLContext class is an encapsulation of a low-level, system-dependent handle into the internal OpenGL structure containing full state of a specific rendering device, including definitions of display-lists and texture-objects. OpenGL allows sharing of these resources among several windows, which is exactly what we need to provide for sharing of scene object-data among several viewers. Additional class `TGLContextIdentity` is used to facilitate management of concurrent GL contexts.

[4] This also allows for inclusion of 2D graphics elements and the necessary extensions are in development.

5. Direct OpenGL rendering

The idea behind direct OpenGL rendering is to allow users to directly inject OpenGL code into the scene rendering pipeline. This is achieved by sub-classing of the `TGLObject` class and implementing two virtual functions. For example, imagine we have a ROOT class `TPointSet` and we want to provide a special rendering of its data. We have to implement a `TPointSetGL` class (name of the original class post-fixed with GL) with the following functions:

```
class TPointSetGL : public TGLObject
{
  virtual Bool_t SetModel(TObject* obj);
  virtual void   DirectDraw(TGLRnrCtx& ctx);
};
```

The `TPointSet::Paint()` function now only fills the core sections of the `TBuffer3D` structure: pointer to itself, transformation matrix and default color of the object. The scene infrastructure automatically searches the class dictionary for a GL-postfixed version of the original class' name.[5] When a match is found, the GL-object is instantiated and the `SetModel()` function is called to pass the original object to the renderer.

The `DirectDraw()` function is called during each render-pass and now the GL object can directly access the data of its creator and call arbitrary OpenGL functions. If one modifies the OpenGL state variables, they should be restored to their previous values. Examples of direct-rendering classes used for raw-data visualization in ALICE are shown in Fig.2.

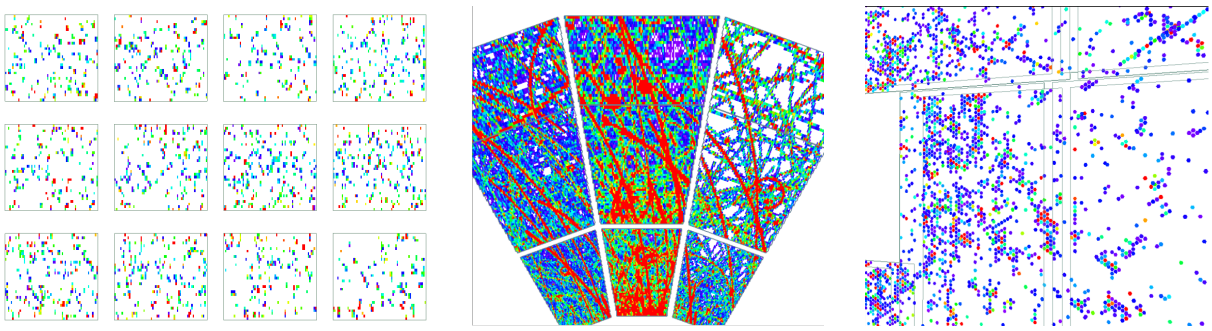

Figure 2. Examples of raw-data rendering in ALICE Event Visualization Environment using the direct-rendering infrastructure: inner tracking system (left), time projection chamber (middle) and photon-multiplicity detector (right).

Besides removing all the restrictions on the visual representation of ROOT objects, the described mechanism also avoids copying of data from the data-object into the `TBuffer3D` and from it to the renderer (and then, once more, into the graphics-card memory). This is important for large objects especially if they need to be updated frequently.

Our experience with direct-rendering shows, that a lot of flexibility can be achieved with a small number of rendering-classes. For example, in AliEVE a single rendering class is used for raw-data rendering of nine different sub-detectors.

6. Two-level selection

Two-level selection provides a framework-wide solution for interaction with sub-elements of rendering objects. Imagine a collection of clusters, where one wants to interact both with the

[5] If this fails, all base-classes are searched as well. Afterwards the corresponding `TClass` pointers are stored into a map so that the relatively expensive dictionary look-up can be avoided.

collection as a whole (to move or delete it, to change palette mapping or cuts, etc.) as well as to be able to obtain detailed information about an individual cluster.

OpenGL provides a mechanism for tagging of visual representations and a special selection mode, which returns the tags of objects that intersect given screen area, usually a small square around the mouse position. This is used by the ROOT GL viewer to perform object selection and manipulation on the level of rendering objects.

When rendering a large collection of objects one uses, for performance reasons, a set of special GL commands that allow many graphics primitives to be rendered with a single function call. In this mode, however, it is not possible to tag each sub-element. To maintain maximum performance and still be able to perform detailed selections, we perform such operations in two steps.

(i) The first-pass selection and on-screen rendering are done with the optimized, bulk-processing commands. The result of the first-pass selection is a single object.

(ii) During the second-pass this single object is rendered along a special code-path where each sub-element is appropriately tagged.

Thus we retain performance and flexibility at the expense of writing two rendering code segments for each container object that needs to provide introspection into its elements.

To use this functionality, one needs to implement two more virtual functions of the `TGLObject` class:

```
class TPointSet3D : public TGLObject
{
  virtual Bool_t SupportsSecondarySelect() { return kTRUE; }
  virtual void   ProcessSelection(TGLSelectRecord& rec);
};
```

The first one simply states that this class supports two-level selection. The second one is called by the viewer after the secondary selection is finished and the result needs to be processed. It is a natural obligation of the rendering-class to demangle the selection-record as it is the only one knowing how the sub-elements were tagged. From here, further calls to functions in the original, data-holding object can be made.

As an example consider a 2D representation of an ALICE TPC sector where a color-coded rectangle is displayed for each pad. In optimized mode, the whole sector is rendered as three textured rectangles, spending about 20 calls to OpenGL. In the second-pass of the selection it is rendered as 8,000 rectangles, costing us approximately 7*8,000 OpenGL calls.[6] After the secondary hit is registered the `TPCSector2D::PadSelected(row, pad)` function is called in the original object. Depending on its mode, it prints out the pad details or displays a signal versus time histogram of the pad or of the entire pad-row.

7. Overlay event management

While restructuring the implementation of object and clipping-plane manipulators it turned out that the cleanest way to separate them from the viewer is to provide a general infrastructure for handling of GUI events within a GL window and deliver them to specific objects that perform arbitrary actions based on this input. Thus, *overlay* is a set of objects, registered to a specific viewer, that are able to render themselves and handle GUI events. Viewer plays a role similar to the X11 server: it detects which element is currently under the mouse pointer and forwards to it all events that it receives.

To use it, one has to sub-class the `TGLOverlayElement` class:

[6] Six for rectangle rendering, one for pad-tag registration.

```
class TGLOverlayElement
{
 virtual Bool_t MouseEnter(TGLOvlSelectRecord& selRec);
 virtual Bool_t Handle(TGLRnrCtx& rnrCtx, TGLOvlSelectRecord& selRec, Event_t* event);
 virtual void   MouseLeave();
 virtual void   Render(TGLRnrCtx& rnrCtx) = 0;
};
```

`MouseEnter()` and `MouseLeave()` functions should be reimplemented if the overlay element has some inactive parts. After the element returns *true* to the `MouseEnter()` offer, `Handle()` will be called on every event (including keyboard events) received by the viewer, until the mouse leaves the said element. The `Render()` function can make any OpenGL calls and should use selection-tagging commands to describe its sub-elements. Overlay elements are not limited to flat 2D representations and can also use existing scene-elements to calculate position, orientation and scale of their visual representations.

This allows inclusion of arbitrary, user-provided interaction objects which can respond dynamically to the user-input and change the state of scene-objects, of the viewer or of the application itself. Further, one can also implement OpenGL widgets, like buttons or scroll-bars.

8. Conclusion

We have presented the major advancements in the OpenGL support within the ROOT framework introduced in the last year. The existing implementation has been modularized and extended, allowing users to exert more control on all levels of the infrastructure as well as to reuse the provided functionality in variety of contexts. The developments were mostly driven by the needs of the ALICE collaboration which requires, due to large data-size of lead–lead events, a flexible and fully interactive visualization environment. The code described in this article is available in ROOT since v5-16/00.

Additionally, the experiment-independent part of ALICE Event Visualization Framework is planned to be included in the ROOT distribution before the next production release, v5-18/00. This will expose the new functionality to users of all levels and we believe that user-feedback will allow us to further enrich the framework with all the elements interesting for the high-energy physics community.

Acknowledgments

The author would like to thank René Brun, Alja Mrak–Tadel and Timur Pocheptsov for their help and support.

References

[1] Brun R and Rademakers F 1997 *Nucl. Inst. & Meth. in Phys. Res.* A **389** pp 81-86.
[2] For current status of the ROOT framework see the project web-page `http://root.cern.ch/` and other ROOT-related contributions to the CHEP-2007 conference.
[3] Couet O, Maunder R, Pocheptsov T and Brun R 2007 *XV Int. Conf. on Comp. in High Energy and Nucl. Phys. 2006* vol 1 (Mumbai: Macmillan) pp 324–327.
[4] Tadel M and Mrak-Tadel A 2007 *XV Int. Conf. on Comp. in High Energy and Nucl. Phys. 2006* vol 1 (Mumbai: Macmillan) pp 398–401.