Available online at www.sciencedirect.com
ScienceDirect
Computer Physics Communications 178 (2008) 331–365
Computer Physics Communications
www.elsevier.com/locate/cpc

The GEANT4 Visualisation System

J. Allison [a,*], M. Asai [b], G. Barrand [c], M. Donszelmann [b], K. Minamimoto [d], J. Perl [b], S. Tanaka [e], E. Tcherniaev [f], J. Tinslay [b]

[a] *University of Manchester, UK*
[b] *Stanford Linear Accelerator Center (SLAC), USA*
[c] *IN2P3/LAL, Orsay, France*
[d] *Institute for Computational Fluid Dynamics, Tokyo, Japan*
[e] *Ritsumeikan University, Japan*
[f] *European Organization for Nuclear Research (CERN), Switzerland*

Received 19 April 2007; received in revised form 31 August 2007; accepted 21 September 2007

Available online 7 October 2007

Abstract

The GEANT4 Visualisation System is a multi-driver graphics system designed to serve the GEANT4 Simulation Toolkit. It is aimed at the visualisation of GEANT4 data, primarily detector descriptions and simulated particle trajectories and hits. It can handle a variety of graphical technologies simultaneously and interchangeably, allowing the user to choose the visual representation most appropriate to requirements. It conforms to the low-level GEANT4 abstract graphical user interfaces and introduces new abstract classes from which the various drivers are derived and that can be straightforwardly extended, for example, by the addition of a new driver. It makes use of an extendable class library of models and filters for data representation and selection. The GEANT4 Visualisation System supports a rich set of interactive commands based on the GEANT4 command system. It is included in the GEANT4 code distribution and maintained and documented like other components of GEANT4.

PACS: 07.05.Tp; 13; 23; 89.20.Ff

Keywords: Simulation; Particle interactions; Geometrical modelling; Graphics; Visualisation; Ray tracing; DAWN; HepRep; OpenGL; Open inventor; VRML; Software engineering; Object-oriented technology; Distributed software development

1. Introduction

The GEANT4 Visualisation System is a component of the GEANT4 Simulation Toolkit [1], a body of C++ code that incorporates a vast amount of knowledge about the interaction of particles with matter, used in a wide variety of applications, including the simulation of particle physics experiments, medical physics and space vehicle design. It was always intended that the GEANT4 code distribution should include a visualisation system whose primary purpose was to facilitate the application developer in designing applications and studying the simulated particle interactions. To that end, two basic interfaces were designed, together known as the Graphics Interface. The GEANT4 Visualisation System is an extensive and versatile system comprising a set of classes implementing this interface.

It should be stressed that GEANT4 may be linked with any visualisation system that complies with the Graphics Interface (described below), and sophisticated developers may provide their own in their own project's software framework. To assist this, a whole set of models, for geometrical volumes, axes, text, trajectories, etc., has been written using only the basic graphics interface;

* Corresponding author. Address: School of Physics and Astronomy, The University of Manchester, MANCHESTER M13 9PL, UK. Tel.: +44 161 275 4170; fax: +44 161 273 5867.
E-mail address: John.Allison@manchester.ac.uk (J. Allison).

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2007.09.010

the models, therefore, may be re-used with any visualisation system. GEANT4 may also be linked without a visualisation system at all, for example, for batch processing. In that case, if code remains that uses the Graphics Interface, care must be taken that the thread of control does not pass through it. A simple way of doing that for users is recommended and described below.

On the other hand, most users will wish to use the provided visualisation system, at least at some time in the development of their application. It is this provided system that we refer to as *the Geant4 Visualisation System*. We hope its features will make it attractive to the user. It supports multiple simultaneous graphics technologies each with several variants—20 distinct drivers at the last count—ranging through: dumb terminal output; immediate non-interactive displays; highly interactive displays with their own graphical database; off-line browsers; pipe-connected or network-connected (sockets) displays; with a variety of features which sometimes focus on a particular quality such as: interactivity; speed; quality of printed output; perhaps highlighting different aspects of simulation, such as: the geometry shapes; geometry hierarchy; particle tracks and interactions. Moreover, the user can switch graphics drivers without losing information about the content, style or viewpoint of the view. Utilities are provided that can decompose the more complex solids and primitives that advanced drivers may handle as native objects into simpler primitives for less advanced drivers. Appendix A lists the currently available graphics drivers.

The features of the GEANT4 Visualisation System are not fixed. They are evolving with time in response to user demand. Activity in the development of the GEANT4 Visualisation System is still at a high level and new features are being continuously added. However, it has reached a significant level of maturity to an extent that was thought to justify documenting its design at this time. The basic design is unlikely to change. Nevertheless, this paper represents the current snapshot and the user should keep up with GEANT4's release notes. In particular, a user can obtain a good impression of its features by browsing the interactive commands either within an interactive application or as described in the User Guide for Application Developers [2]. A good introductory presentation is maintained at the SLAC web site [3]. The snapshot described in this paper corresponds to GEANT4 Version 9.0 (June 2007).

2. Structure of the paper

The aim is to make a technically accurate and complete description of the graphical components of GEANT4 for readers interested in its design and for developers requiring to understand its workings. It is not a user manual; for that, the user should consult the GEANT4 User Guides accessible from the GEANT4 web page [1] or the introductory presentation [3]. Nevertheless, Sections 4 (Graphical representations) and 6 (The `G4VVisManager` user interface) are relevant to application programming, while Sections 8 (The GEANT4 Visualisation System), 9 (Visualisation commands) and 10 (Available graphics drivers) give a good insight into the GEANT4 Visualisation System's philosophy and capabilities. The average GEANT4 user might find it better to skip Sections 5 and 7, which are more relevant to a developer of the visualisation system.

Description proceeds in a logical order, by which we mean later sections refer to earlier ones in a use-relationship that corresponds closely to the GEANT4 category dependencies. After an Overview, we begin with a description of the low-level graphical representations category, which includes the basic graphical objects that are ultimately processed by the graphics drivers. Next, we describe the two main interfaces, `G4VGraphicsScene` and `G4VVisManager`, that are available to the GEANT4 programmer. The former is for GEANT4 toolkit developers only. The latter would typically be used by an application developer writing code to draw hits.

Section 7 introduces the concept of models and filters. Only in Section 8 the GEANT4 Visualisation System is itself finally described. Section 9 outlines the all-important visualisation commands. Finally, the currently available graphics drivers are showcased in Section 10.

Many details are relegated to extensive appendices.

3. Overview

The GEANT4 Visualisation System consists of the GEANT4 Visualisation Manager, `G4VisManager`, associated managers, so-called scene handlers that convert the scene of GEANT4 objects into graphics system-specific data, and viewers that actually render a scene to a window or file. Fig. 1 shows the class diagram, including the abstract bases classes from which they are derived. The user may instantiate any number of scenes and attach them to any number of various types of scene handler. Thus various scenes can be viewed in various ways. At any given time only one viewer is active, for actions such as drawing trajectories or changing viewpoint, but a change of scene triggers the refreshing of all views of that scene.

The user typically interacts with the system through commands such as `/vis/scene/create`—see Section 9. The user may also be called upon to write code to represent hits in terms of the basic graphical representations (Section 4). In fact, he or she may write code to draw anything to the current view at any time, for example, at the end of event or even at the end of each tracking step. If the visualisation system's effect on tracking performance is an issue, it may be disabled so that drawing is bypassed and enabled again when desired.

The "workhorse" of the GEANT4 Visualisation System is OpenGL. Nearly all computers are delivered with this already installed. It gives fast, immediate graphics and its graphical database (display lists) is exploited for rapid view manipulation. But other drivers

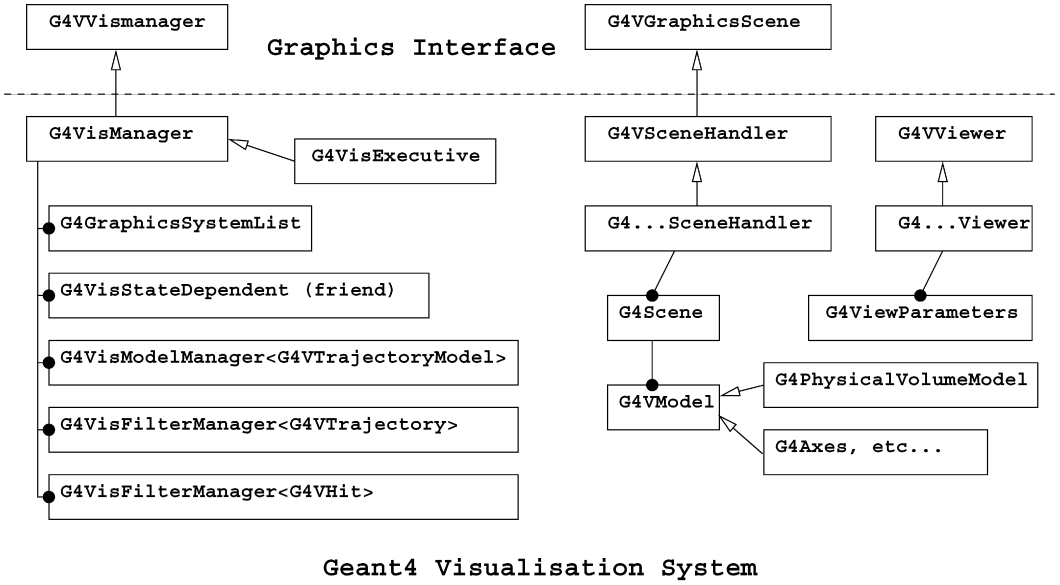

Fig. 1. The GEANT4 Visualisation System class diagram. `G4GraphicsSystemList` contains the factories for the scene handlers and viewers. `G4Scene` holds scene models. The `G4VisModelManager` and the `G4VisFilterManagers` create and manage trajectory and hit models and/or filters.

are available—see Section 10—for producing high quality PostScript or interfacing to highly interactive browsers. There is even one driver that uses GEANT4's own tracking algorithms to produce photo-realistic images—this is also useful for testing the tracking algorithms and, by comparing with other viewers, for ensuring that they accurately represent the geometry that GEANT4 is navigating.

4. Graphical representations

This low-level category provides some basic graphical representations of common GEANT4 objects—`G4Circle`, `G4Polyhedron`, `G4Polyline`, `G4Polymarker`, `G4Scale`, `G4Square` and `G4Text`—the so-called graphics primitives. `G4Polyhedron` has constructors for most geometry solids; those that do not must use a general constructor. `G4Polyhedron` also supports so-called Boolean operations—union (addition), intersection and subtraction—which allows polyhedral representation of the equivalent GEANT4 solids, `G4UnionSolid`, `G4IntersectionSolid` and `G4SubtractionSolid`.

This category also includes `G4AttDef` and `G4AttValue`, C++ implementations of HepRep [4] objects that may be used to pass generic information to the visualisation system. They and their use are described in Appendix J.

The basic graphical representations inherit `G4Visible` which has a `G4VisAttributes` pointer. `G4VisAttributes` may be similarly attached to geometry volumes through `G4LogicalVolume`. This is shown in the class diagram of Fig. 2. Attributes that may be assigned in this way include: colour, drawing style, visibility and `G4AttDef/Values`—see Table 1. Note the flags and parameters for forcing drawing style, etc. If set, those attributes are expected to take precedence over the current defaults in the graphics system. "Auxiliary edges" refer to polygon edges of a polyhedral representation that are in a surface (curved or planar); they are not real edges and would normally not be drawn. Set this, for example, of you wish to ensure that a sphere is visible in wireframe mode. Similarly, the precision with which a circular edge can be specified by the number of polygon edges. This is recommended for volumes containing only a small angle of circle, for example, a thin tube segment. The User Guide for Application Developers [2], Section 8.3, Visualisation Attributes, describes their use in detail. Issues surrounding the use of `G4VisAttributes` for the toolkit developer are addressed in Appendix C.

5. The `G4VGraphicsScene` low-level interface (also known as the scene handler interface)

This is intended only for toolkit developers or users who provide their own visualisation system; it is used only by (a) the geometry category for describing shapes and (b) by models (see below). A good impression of the functionality of the interface can be gained from a comment-stripped C++ class definition shown in Fig. 3. It is dominated by `Add...` methods for (a) some specific solids and a general solid, (b) two special (compound) objects (trajectory and hit), and (c) the graphics primitives. Details of its use are described in Appendix B. Its place in the class hierarchy is shown in Fig. 1.

Concrete instances of this class are called *scene handlers* in the parlance of the GEANT4 Visualisation System. Each scene handler is specific to the underlying graphics technology.

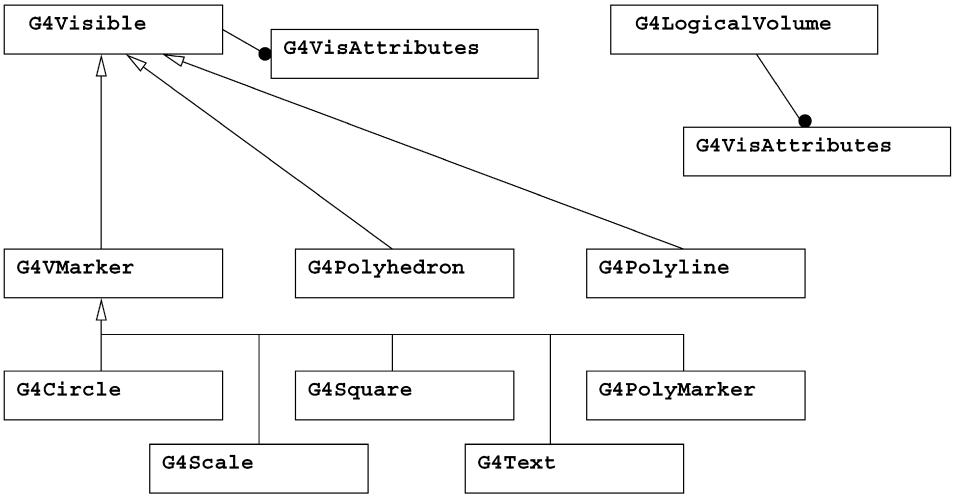

Fig. 2. The class diagram of graphics primitives and related classes, showing their relationship to `G4VisAttributes`.

Table 1
Visualisation attributes (`G4VisAttributes`)

`G4bool fVisible;`	Visibility flag
`G4bool fDaughtersInvisible;`	Make daughters invisible.
`G4Colour fColour;`	
`LineStyle fLineStyle;`	
`G4double fLineWidth;`	Units of "normal" device line width, e.g., pixels for screen, 0.1 mm for paper.
`G4bool fForceDrawingStyle;`	To switch on forced drawing style.
`ForcedDrawingStyle fForcedStyle;`	Value—wireframe/surface/etc.
`G4bool fForceAuxEdgeVisible;`	Force drawing of auxiliary edges.
`G4int fForcedLineSegmentsPerCircle;`	Forced lines segments per circle.
`G4double fStartTime, fEndTime;`	Time range.
`const std::vector<G4AttValue>* fAttValues`	For picking, etc.
`const std::map<G4String,G4AttDef>* fAttDefs`	Corresponding definitions.

6. The `G4VVisManager` user interface

The first thing to notice—see Fig. 4—is the static method `GetConcreteInstance`. This returns zero unless the concrete object has been instantiated and is available. The user must protect drawing code as follows:

```
G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
if(pVVisManager) {
  pVVisManager->Draw(circle);
  ...
```

so that the thread of control does not pass through the interface if the visualisation manager does not exist or is not ready. (It may come as a surprise that one may link an application without a concrete implementation of a used interface; it is indeed possible, but, in that case, the user must ensure that the thread of control does not pass through.) All drawing code in the GEANT4 examples is protected like this. The novice examples can be built and run without a concrete visualisation manager by setting the environment variable `G4VIS_NONE`.

This interface consists largely of `Draw` methods for basic graphical representations. A typical concrete visualisation manager would hand the drawing to the scene handler:

```
void ConcreteVisManager::Draw
 (const G4Circle& circle,
  const G4Transform3D& objectTransformation) {
  sceneHandler->BeginPrimitives(objectTransformation);
  sceneHandler->AddPrimitive(circle);
  sceneHandler->EndPrimitives();
}
```

```
class G4VGraphicsScene {
public:
  virtual void PreAddSolid (const G4Transform3D& objectTransformation,
                            const G4VisAttributes& visAttribs) = 0;
  virtual void PostAddSolid () = 0;
  virtual void AddSolid (const G4Box&)       = 0;
  virtual void AddSolid (const G4Cons&)      = 0;
  virtual void AddSolid (const G4Tubs&)      = 0;
  virtual void AddSolid (const G4Trd&)       = 0;
  virtual void AddSolid (const G4Trap&)      = 0;
  virtual void AddSolid (const G4Sphere&)    = 0;
  virtual void AddSolid (const G4Para&)      = 0;
  virtual void AddSolid (const G4Torus&)     = 0;
  virtual void AddSolid (const G4Polycone&)  = 0;
  virtual void AddSolid (const G4Polyhedra&) = 0;
  virtual void AddSolid (const G4VSolid&)    = 0;  // For solids not above.
  virtual void AddCompound (const G4VTrajectory&) = 0;
  virtual void AddCompound (const G4VHit&)        = 0;
  virtual void BeginPrimitives
  (const G4Transform3D& objectTransformation = G4Transform3D()) = 0;
  virtual void EndPrimitives () = 0;
  virtual void BeginPrimitives2D () = 0;
  virtual void EndPrimitives2D   () = 0;
  virtual void AddPrimitive (const G4Polyline&)   = 0;
  virtual void AddPrimitive (const G4Scale&)      = 0;
  virtual void AddPrimitive (const G4Text&)       = 0;
  virtual void AddPrimitive (const G4Circle&)     = 0;
  virtual void AddPrimitive (const G4Square&)     = 0;
  virtual void AddPrimitive (const G4Polymarker&) = 0;
  virtual void AddPrimitive (const G4Polyhedron&) = 0;
};
```

Fig. 3. The `G4VGraphicsScene` low-level interface.

```
class G4VVisManager {
public:
  static G4VVisManager* GetConcreteInstance ();
  virtual void Draw (const G4Circle&,
    const G4Transform3D& objectTransformation = G4Transform3D()) = 0;
  virtual void Draw (const G4Polyhedron&,...  //...as above.
  virtual void Draw (const G4Polyline&,...
  virtual void Draw (const G4Polymarker&,...
  virtual void Draw (const G4Scale&,...
  virtual void Draw (const G4Square&,...
  virtual void Draw (const G4Text&,...
  virtual void Draw2D (const G4Text&) = 0;
  virtual void Draw (const G4VHit&) = 0;
  virtual void Draw (const G4VTrajectory&, G4int i_mode = 0) = 0;
  virtual void Draw (const G4LogicalVolume&, const G4VisAttributes&,...
  virtual void Draw (const G4VPhysicalVolume&, const G4VisAttributes&,...
  virtual void Draw (const G4VSolid&, const G4VisAttributes&,...
  virtual void GeometryHasChanged () = 0;
  virtual void DispatchToModel (const G4VTrajectory&, G4int i_mode = 0) = 0;
  virtual G4bool FilterTrajectory(const G4VTrajectory&) = 0;
  virtual G4bool FilterHit(const G4VHit&) = 0;
protected:
  static void SetConcreteInstance (G4VVisManager*);
  static G4VVisManager* fpConcreteInstance;  // Pointer to real G4VisManager.
};
```

Fig. 4. The `G4VVisManager` user interface.

where *sceneHandler is an object that conforms to the G4VGraphicsScene interface. An important point to note is that this gives the visualisation manager the opportunity of managing multiple scene handlers, which is its reason for existing alongside the G4VGraphicsScene low-level interface.

There are also Draw methods for hits and trajectories. The scene handler has the opportunity of dealing with them or handing them back to the user written drawing methods (G4VHit::Draw and G4VTrajectory::DrawTrajectory), which typically decompose the drawing into basic graphical representations. DispatchToModel is used by the default implementation of G4VTrajectory::DrawTrajectory to hand the drawing of trajectories back again to the visualisation manager.[1] This invocation sequence gives the user total control over the drawing of hits and trajectories and preserves backwards compatibility, while, in the default case, allowing the visualisation manager to select drawing models. This is discussed in detail in Appendix E. FilterTrajectory and FilterHit are intended for use by DispatchToModel, but is offered in this interface for general use. Trajectory models and filters are described in detail in Appendices H and I.

Also there are Draw methods for volumes and solids. Thus geometry solids can be used like basic graphical representations to represent hits, for example. Again, as described above, the scene handler has the opportunity of dealing with them or handing them to objects that decompose them into basic graphical representations. Note that the G4VisAttributes specified in the argument overrides any attributes previously assigned to the volume, so that, for example, the user can record the energy deposited in a sensitive volume and colour it according to energy.

Finally, GeometryHasChanged is invoked by the run manager to inform the visualisation manager that the geometry has changed.

7. Models and filters

Models are objects that decompose the target into basic graphical representations through the Graphics Interface. There are two[2] types:

- **Scene models.** Derived from G4VModel, these are designed to be components of a viewable scene (Section 8.4).
- **Trajectory models.** Derived from G4VTrajectoryModel, these are selected by the visualisation manager and used when trajectory drawing has been requested.

Filters, for both trajectories and hits, are derived directly from G4VFilter or indirectly via G4SmartFilter, which is itself derived from G4VFilter. They allow the visualisation manager to decide, on the basis of trajectory or hit parameters, whether to draw or not.

7.1. Scene models

Most scene models use only the G4VGraphicsScene low-level interface because (a) it is available through the void DescribeYourselfTo(G4VGraphicsScene&) function and (b) it is more efficient, but they may, in fact, use either of the basic graphics interfaces. This relies on the visualisation manager being consistent in its communication with the current scene handler. The rule is: calls to the G4VVisManager user interface must not occur inside a Begin/EndPrimitives sequence, as that would cause nesting of Begin/EndPrimitives sequences, which is not supported.

Table 2 lists current scene models. They inherit the abstract base class G4VModel, which has textual descriptions, extent and position/orientation (transform) and, in particular, a pointer to modelling parameters (Table 3), if required. Modelling parameters are used, for example, to communicate culling policy.

The main task of a scene model is to describe itself to the scene handler through the G4VGraphicsScene low-level interface by implementing the virtual method void DescribeYourselfTo(G4VGraphicsScene&). It then lends itself to the concept of a scene, which is an object that is defined by a list of models. This is discussed in Section 8.4.

Of particular mention is G4PhysicalVolumeModel. It models a given physical volume and its daughters to a specified depth. It operates the culling policy defined in the modelling parameters, so that, for example, volumes marked invisible are not drawn. Culling also extends to covered daughters (to economise on the rendering of objects that will never be seen) and density (so that, for example, volumes made of air are removed). The G4PhysicalVolumeModel descends and traverses the geometry hierarchy and selects solids to pass to the scene handler. It keeps a record of: the current depth in the geometry tree relative to the given physical volume; the current physical volume; the current logical volume; and the *drawn* path, i.e. the path of parent–child relationships *excluding* culled volumes. If the scene handler needs any of this information, for example, to obtain textual tags or build its own scene graph (to use Open Inventor parlance), it is necessary to make a dynamic cast:

[1] It is anticipated that a similar mechanism for specialised hits will be introduced in future.

[2] Hits models may be added in future.

Table 2
Models

G4AxesModel	Models the basic graphical representation, G4Axes, at x, y, z with length.
G4CallbackModel	Template class for a user-defined function that will be invoked if the model is added to the scene. This can be done with the G4VUserVisAction facility. See the User Guide for Application Developers [2], Sections 8.8.7 and 8.8.8.
G4HitsModel	Extracts hits from the hits collections of this event.
G4LogicalVolumeModel	Models the volume and immediate daughters in its own coordinate system, with Boolean components, voxels, and readout geometry, if requested and if any.
G4PhysicalVolumeModel	The work-horse of geometry representations—see text.
G4ScaleModel	Models the basic graphical representation, G4Scale.
G4TextModel	Models the basic graphical representation, G4Text.
G4TrajectoriesModel	Extracts trajectories from the trajectory container.

Table 3
Modelling parameters

const G4VisAttributes* fpDefaultVisAttributes;	
DrawingStyle fDrawingStyle;	Wireframe, surface, hidden line, etc.
G4bool fCulling;	Culling requested.
G4bool fCullInvisible;	Cull (do not Draw) invisible objects.
G4bool fDensityCulling;	Density culling requested. If so...
G4double fVisibleDensity;	...density lower than this not drawn.
G4bool fCullCovered;	Cull daughters covered by opaque mothers.
G4double fExplodeFactor;	Explode along radius by this factor...
G4Point3D fExplodeCentre;	...about this centre.
G4int fNoOfSides;	...in polygon circle approximation.
const G4Polyhedron* fpSectionPolyhedron;	For generic section (DCUT).
const G4Polyhedron* fpCutawayPolyhedron;	For generic cutaways.
const G4Event* fpEvent;	Event being processed.

```
  G4PhysicalVolumeModel* pPVModel =
    dynamic_cast<G4PhysicalVolumeModel*>(fpModel);
  if (pPVModel) {
    tag = pPVModel->GetCurrentPV()->GetName();
    ...
  } else {
    // Not from a G4PhysicalVolumeModel.
...
```

Appendix G discusses the building of drawn scene graphs. Note that it is always possible that the solid does *not* come from a G4PhysicalVolumeModel (perhaps it is a volume representing a hit) in which case it must be treated differently, with no place in the geometry hierarchy.

Note also that the above requirement to make a dynamic_cast deals with the possibility that the current object is *not* generated by a model at all—for example, it may come direct from user code, in which case the scene handler must ensure fpModel itself is zero.

7.2. Trajectory models

Trajectory models do not have direct access to the scene handler, so must use the G4VVisManager user interface.

A trajectory model governs how an individual trajectory is drawn. Concrete models inherit the G4VTrajectoryModel interface. They are typically invoked by the visualisation manager when a trajectory requests through DispatchToModel, but may be invoked directly by the user in the Draw method of the trajectory or elsewhere.

A growing set of trajectory models is provided in the code distribution, for example, G4TrajectoryDrawByCharge and G4TrajectoryDrawByParticleID. To facilitate management they are registered via G4VModelFactory described below. They are supported by a command structure. More information is given in Appendix H.

The user may write his or her own trajectory model. As a way of implementing a personal way of drawing trajectories, it is preferred over the old (but still supported) way of writing the DrawTrajectory method of the trajectory. The trajectory model must be registered directly with the GEANT4 Visualisation Manager (Section 8.1) with `G4VisManager::RegisterModel(G4VTrajectoryModel*)`.

7.3. Filters

Filters have a similar design to trajectory models. A good set is provided in the code distribution for both trajectories and hits, registered via `G4VModelFactory` and supported by a command structure. Like trajectory models, the user may write a personal filter, derived from `G4VFilter` or `G4SmartFilter`, and register it with the GEANT4 Visualisation Manager. More information is given in Appendix I.

7.4. *G4VModelFactory*

`G4VModelFactory` is an abstract base class for factories that create models, filters and associated messengers. Concrete instances of this class are intended for use by the visualisation manager in its management of models and filters. (Scene models are handled in a different way, namely as components of `G4Scene`—see Section 8.4.)

`G4VModelFactory` assumes the model has an associated messenger.[3] In this case, there must be a mechanism to generate both models and their associated messengers. This is the role of `G4VModelFactory`. Concrete factories inheriting from `G4VModelFactory` are responsible for creating a concrete model and a concrete messenger. To help ensure a type-safe messenger-to-model relationship, the messengers should inherit `G4VModelCommand`. More information is given in Appendix I.

8. The GEANT4 Visualisation System

We are now in a position to describe the GEANT4 Visualisation System, which makes use of all the components and features of the above sections. It provides visualisation drivers, some of which exploit external graphics technologies.

A wide variety of user requirements went into the design of the GEANT4 Visualisation System, for example (to quote closely from the original documentation [2]):

- very quick response in surveying successive events;
- high-quality output for presentation and documentation;
- flexible camera control for debugging detector geometry and physics;
- selection of visualisable objects;
- interactive picking of graphical objects for attribute editing or feedback to the associated data;
- highlighting incorrect intersections of physical volumes;
- co-working with graphical user interfaces.

Because it is very difficult to respond to all of these requirements with only one built-in visualiser, the Graphics Interface (defined, you will recall, as the two basic interfaces, `G4VGraphicsScene` and `G4VVisManager`), was developed to support multiple graphics drivers over several complementary graphics technologies to satisfy a wide variety of users' needs. Here the term *graphics technology* means either an application running as a process independent of GEANT4 or a graphics library to be compiled with GEANT4. A concrete implementation of the `G4VGraphicsScene` low-level scene-handler interface, together with its viewers, is called a *visualisation driver* or *graphics driver*. This may use a graphics library directly, communicate with an independent process via pipe or socket, or simply write an intermediate file for a separate viewer.

8.1. The GEANT4 Visualisation Manager

`G4VisManager` implements the `G4VVisManager` user interface. It manages multiple graphics drivers and defines three more concepts: the *scene* (`G4Scene`, Section 8.4); the *scene handler* (base class `G4VSceneHandler`, itself a sub-class of `G4VGraphicsScene`, Section 8.5); the *viewer* (base class `G4VViewer`, Section 8.6). Fig. 1 shows the class structure.

Note there is no restriction on the number or type of scene handlers or viewers. There may be several scene handlers processing the same or different scenes, each with several viewers showing, for example, the same scene from differing viewpoints. It maintains a concept of *current graphics driver factory, current scene, current scene handler* and *current viewer*.

[3] A messenger defines and executes user commands. For information about GEANT4 commands and the design of messengers, see the User Guide for Application Developers [2].

G4VisManager is a singleton. It is also an abstract class. The reason for this is that the instantiation of specific drivers must be done in the user domain to avoid circular linking dependencies. Also, only the user knows which graphics technologies are actually available on the computer running the application. To facilitate this, the code distribution includes G4VisExecutive, described in the next section.

A friend class, G4VisStateDependent, notifies G4VisManager at the beginning and end of each run and each event. At the end of event, trajectories, etc., if requested, are drawn.

At the end of a run, the G4VisManager asks the run manager to keep the last event of the run (unless the user has already independently issued a request to keep events), so that trajectories, etc., get redrawn when viewers are refreshed or changed. It will, if requested as an end of event action, ask to keep an arbitrary number of events, which can be accumulated (overlaid) or reviewed one by one at the end of the run.

In its role as a manager, G4VisManager naturally provides several administrative and access functions that could, in principle, be used to program visualisation actions. However, a great deal of intelligence has been programmed into the built-in visualisation commands, described in Section 9, and it is strongly recommended that users interact with the GEANT4 Visualisation Manager via these commands. This can be done in code, for example:

```
G4UImanager::GetUIpointer()->ApplyCommand("/vis/viewer/refresh");
```

or, more usually, through the command line of the (graphical) user interface. (A typical GEANT4 application is script driven; it is straightforward to add application-specific commands to the built-in command set.)

The GEANT4 Visualisation Manager also operates a verbosity policy based on a simple 7 level graded system:

```
enum Verbosity {
  quiet,         // Nothing is printed.
  startup,       // Startup and endup messages are printed...
  errors,        // ...and errors...
  warnings,      // ...and warnings...
  confirmations, // ...and confirming messages...
  parameters,    // ...and parameters of scenes and views...
  all            // ...and everything available.
};
```

This can be established at construction or with the command /vis/verbose, which responds to a digit or the first letter of the enum names (q,s, ...). By default, the verbosity level is warnings, but a novice user is recommended to specify confirmations, which prints comforting messages.

8.2. G4VisExecutive

G4VisExecutive is used in all GEANT4 examples. It implements two virtual functions, RegisterGraphicsSystems and RegisterModelFactories. It is implemented as a .hh-.icc combination that is designed to be included in the users' code:

```
#ifdef G4VIS_USE
#include "G4VisExecutive.hh"
#endif
  ...
#ifdef G4VIS_USE
  G4VisManager* visManager = new G4VisExecutive;
  visManager->Initialise();
#endif
  ...
#ifdef G4VIS_USE
  delete visManager;
#endif
```

Initialise calls RegisterGraphicsSystems and RegisterModelFactories.

A typical fragment of code from RegisterGraphicsSystems is:

```
#ifdef G4VIS_USE_OPENGLX
  RegisterGraphicsSystem(new G4OpenGLImmediateX);
```

```
  RegisterGraphicsSystem(new G4OpenGLStoredX);
#endif
```

and from `RegisterModelFactories`:

```
  RegisterModelFactory(new G4TrajectoryDrawByChargeFactory);
  RegisterModelFactory(new G4TrajectoryDrawByParticleIDFactory);
```

Notice the use of C-pre-processor macros. The standard make files set these macros if the corresponding environment variables are set. See Appendix A for a list of currently supported drivers and environment variables needed, if any. Note that the `Configure` script in the standard code distribution sets these environment variables on response to simple questions. Thus the user can control which graphics drivers are linked and instantiated. If any are set, the standard make files set `G4VIS_USE` (it should not be set by the user). If `G4VIS_NONE` is set, `G4VIS_USE` is not set, so the above code would build an application without a visualisation system.

8.3. Graphics driver factories

The graphics driver is defined by its scene handler and viewer. The GEANT4 Visualisation Manager instantiates scene handlers and viewers via "factories", which are sub-classes of `G4VGraphicsSystem`. It is objects of this class that are the argument of `RegisterGraphicsSystem` described in the previous section. Each such object is a factory for graphics-technology-dependent scene handlers and viewers. Details for the toolkit developer are given in Section D.2.

8.4. Scenes

Objects of class `G4Scene` consist of a list of scene models for physical volumes, axes, hits, trajectories, etc. (scene models are described in Section 7). Scene models are distinguished by their lifetime: *run-duration* for physical volumes, axes, etc.; *end-of-event* for hits, trajectories, etc. For example, the visualisation manager selects end-of-event models for processing when notified by the state manager, via `G4VisStateDependent`, that event processing is complete.

The scene has an *extent* (`G4VisExtent`), which is updated by the scene when a new model is added (each model itself has an extent), and a "standard" target point; these are used to define the standard view—see Section 8.6. In addition, the scene keeps flags which indicate whether end-of-event objects should be accumulated or refreshed for each event or run.

8.5. Scene handlers

`G4VSceneHandler` implements the `G4VGraphicsScene` low-level interface and is itself an abstract base class for a specific scene handler, whose job is to process the scene, i.e. convert it into graphics-technology-specific code for its viewers.

Each scene handler has an attached scene, as shown in Fig. 1. When a viewer is required to render a view, it asks the scene handler to process the scene. In the parlance of scene handlers, this is called a *kernel visit*, since it involves querying objects, such as volumes and trajectories, that are in the Geant4 kernel. By maintaining a simple flag, `fReadyForTransients`, the `G4VSceneHandler` base class distinguishes between *permanent* objects, i.e. those generated by run-duration models and *transient* objects generated by end-of-event models or anything that the user draws via the `G4VVisManager` user interface.[4]

A scene handler may maintain a graphical database, so that a simple change of view parameters, such as a change of viewpoint, does not require a kernel visit. A driver without a graphical database will need to trigger a kernel visit for any change of view parameters. It is the viewer's job to decide when a kernel visit is needed—see Section 8.6.

When a scene handler uses a graphical database, it is required to distinguish between permanent and transient information, so that the visualisation manager may, at end of event, for example, clear and remake just the transient part. Thus the event may be changed without having to redraw the detector. This is described further in Appendix F.

`G4VSceneHandler` imposes some minor additional rules on the concrete sub-classes. These are described in Section D.3.

8.6. Viewers

`G4VViewer` defines the abstract base class for specific viewers. Their job is to create windows or files and identify where and how the final view should be rendered. Each has *view parameters* (`G4ViewParameters`, Section 8.7) which specify viewpoint

[4] Unless as a user action, as described in Section 8.8.

Table 4
Some circumstances that may trigger a kernel visit for drivers with or without a graphical database

Circumstance	With/without graphical database	
	With	Without
`/vis/viewer/refresh`	no	yes
Next event	no (but event gets refreshed by vis manager)	yes
Window exposure	no	yes
Simple change of view params	no	yes
Drastic change of view params	yes	yes
`/vis/viewer/rebuild`	yes	yes

direction, type of rendering (wireframe or surface), etc. It is the viewer's responsibility, noting the scene's extent and target point, to choose a camera position and magnification that ensures that the scene is automatically and comfortably rendered in the viewing window. This is then the *standard view*, and any further operations requested by the user—zoom, pan, etc.—are relative to this standard view. The class `G4ViewParameters` has utility routines to assist this procedure; it is strongly advised that toolkit developers writing a viewer should study the `G4ViewParameters` class, whose header file contains much useful information (also preserved in the Software Reference Manual [5]).

The viewer is messaged by the visualisation manager when the user issues commands, such as `/vis/viewer/refresh`. This invokes methods such as `SetView`, `ClearView` and `DrawView`. A detailed description of the call sequences is given in Appendix E.

As mentioned above, it is the viewer's job to trigger a kernel visit. This is done in `DrawView`; details are given in Sections D.4 and E.1. For file-writing viewers, a kernel visit is preceded by a file re-wind or re-write. Some circumstances that may trigger a kernel visit are shown in Table 4.

8.7. View parameters

View parameters, such as camera direction, drawing styles (wireframe/surface), etc., are held by `G4ViewParameters`. Each viewer holds a view parameters object and a default object (for use in the `/vis/viewer/reset` command). The view parameters can be set for each viewer independently by interactive commands as described below. They are quite extensive—38 at the current count—and a full list is given in Fig. D.3.

8.8. User actions

It is possible to write a sub-class of `G4VUserVisAction` and register it with the GEANT4 Visualisation Manager with `SetUserAction`. It is activated by adding a call-back model to the run-duration list of the scene with the command `/vis/scene/add/userAction`, thereby promoting `Draw` messages to permanent status. A typical use would be to draw a application-specific logo that would be automatically re-drawn as required.

9. Visualisation commands

The GEANT4 Visualisation Manager is controlled by GEANT4 commands. There are over 100 currently available in the `/vis/` command directory and new ones appear continually as new features are added. The user may keep up to date and see the detailed specifications simply by browsing with `help` or `ls` or accessing the User Guide for Application Developers [2], Section 7.1, Built-in commands.

The visualisation commands separate into several broad categories. Here is a selection (without parameters) to indicate the breadth available:

- **Global.** These control overall behaviour.

```
/vis/enable
/vis/disable
/vis/verbose
/vis/reviewKeptEvents
```

- **Compound.** These simply invoke other commands. They represent commonly occurring situations

```
/vis/drawTree
```

```
/vis/drawView
/vis/drawVolume
/vis/open
/vis/specify
```

- **Scene.** Create scenes, control actions, add models (geometry volumes, axes, trajectories, etc.).

```
/vis/scene/create
/vis/scene/endOfEventAction
/vis/scene/endOfRunAction
/vis/scene/list
/vis/scene/notifyHandlers
/vis/scene/select
/vis/scene/add/axes (hits, logo, text, trajectories, volume,...)
```

- **Scene handler.** Create graphics driver-specific scene handlers and attach scenes.

```
/vis/sceneHandler/attach
/vis/sceneHandler/create
/vis/sceneHandler/list
/vis/sceneHandler/select
```

- **Viewer.** A large number of (nearly 40) commands to control the appearance of views. For example:

```
/vis/viewer/clear
/vis/viewer/create
/vis/viewer/refresh
/vis/viewer/zoom
/vis/viewer/set/style
/vis/viewer/set/viewpointThetaPhi
...
```

- **Geometry.** These allow the user to change the visualisation attributes of geometry volumes interactively, for example:

```
/vis/geometry/set/colour
...
```

- **Driver specific.** Some graphics drivers have their own additional commands in the following directories:

```
/vis/ASCIITree/
/vis/heprep/
/vis/rayTracer/
/vis/ogl...
```

- **Modelling and filtering.** These control the display of models, for example:

```
/vis/modelling/trajectories/create/drawByCharge
/vis/modeling/trajectories/drawByCharge-0/set
/vis/filtering/trajectories/create/particleFilter
/vis/filtering/trajectories/particleFilter-0/add
...
```

10. Available graphics drivers

The following graphics drivers are currently included in the GEANT4 code distribution. Table 5 summarises their features. See Appendix A for details about how to invoke.

10.1. The DAWN family

Two drivers exploit the DAWN browser [10], a high quality renderer, including hidden line and hidden surface removal, that produces a PostScript file for publication quality papers and presentations. The first, DAWN, communicates with the DAWN

Table 5
Features of the graphics drivers available in the Geant4 code distribution

Driver family	Advantages	Disadvantages
DAWN	High quality PostScript; sophisticated rendering algorithms.	(Separate) rendering process can be CPU intensive.
HepRep	(With good browser) highly interactive; able to modify and/or select geometry components, tracks and hits.	Wireframe only (no photorealistic rendering).
OpenGL	Fast, immediate graphics; picking of geometry and trajectory information.	Limited to pixel resolution.
OpenInventor	Excellent view manipulation; geometry selection; picking of geometry and trajectory information.	Library installation.
RayTracer	Photorealistic rendering using Geant4 tracking.	CPU intensive.
Tree	Simple and fast way of seeing the geometry hierarchy.	Text only.
VRML	Excellent view manipulation.	Fixed scene.

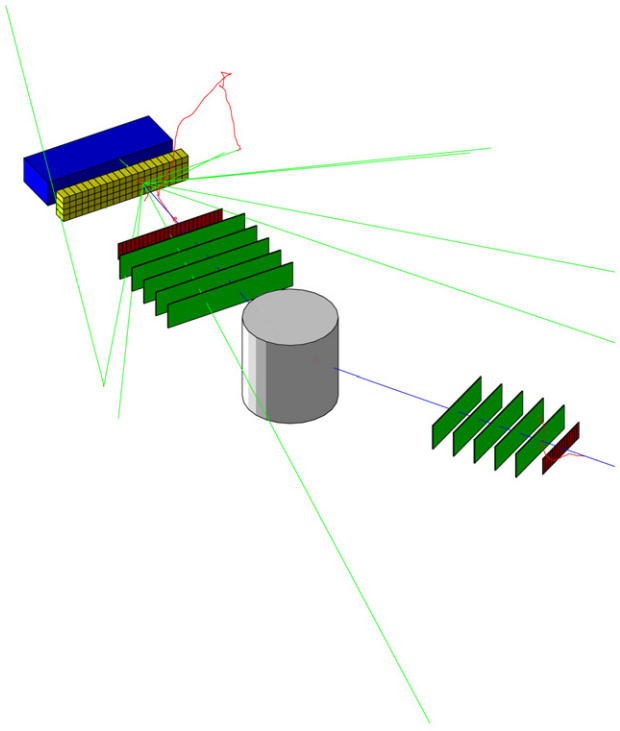

Fig. 5. DAWN view of demonstration detector with trajectories.

browser by USB sockets; the browser must be on standby as a daemon, either locally or remotely. This is a good way of doing remote browsing. The second, DAWNFILE, writes to file, as HepRep (below). A new file is written for every invocation of `ShowView`, which is on the issue of `/vis/viewer/update` or, depending on the scene actions, at the end of each event or run. The DAWN browsers may be invoked automatically in such cases by making "dawn" available in the PATH (Unix parlance) and Fig. 5 is a typical result.

10.2. The HepRep family

These two drivers are typical of file writing drivers. Information is written to file in the HepRep format [4] suitable for browsing by a number of browsers, notably HepRApp [7], WIRED4 [8] and FRED [9]. A new file is written for every invocation of `ShowView`, which is on the issue of `/vis/viewer/update` or, depending on the scene actions, at the end of each event or run.

Figs. 6 and 7 show typical windows opened by the HepRApp browser. It offers many useful features, including:

- picking items and displaying associated information (attributes);
- selecting items by the value of any attribute (cutting);

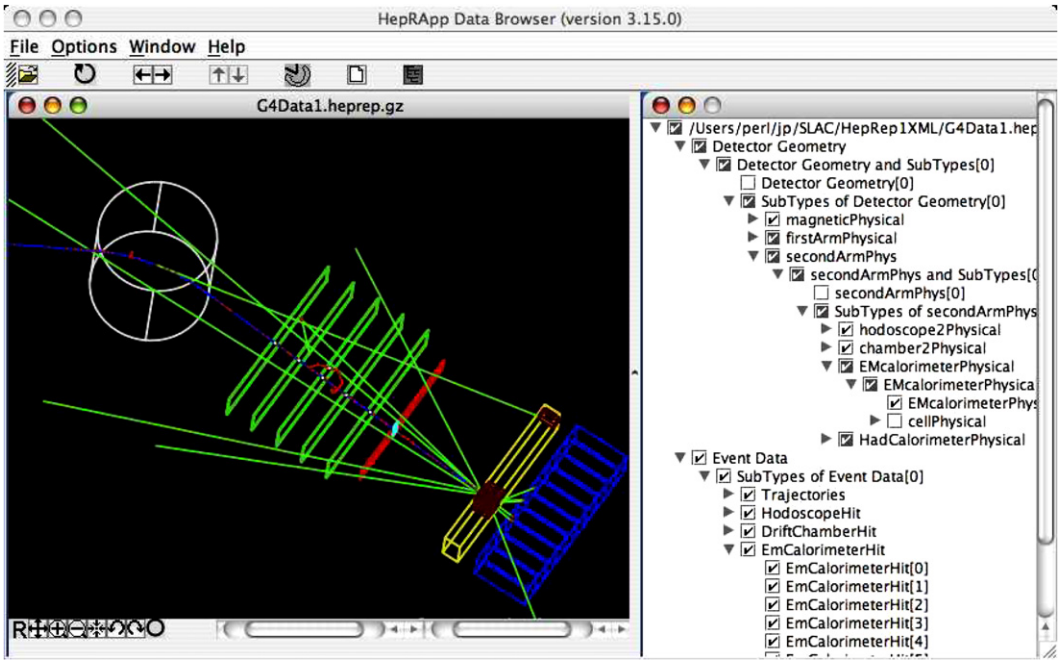

Fig. 6. A HepRApp view of a HepRep file.

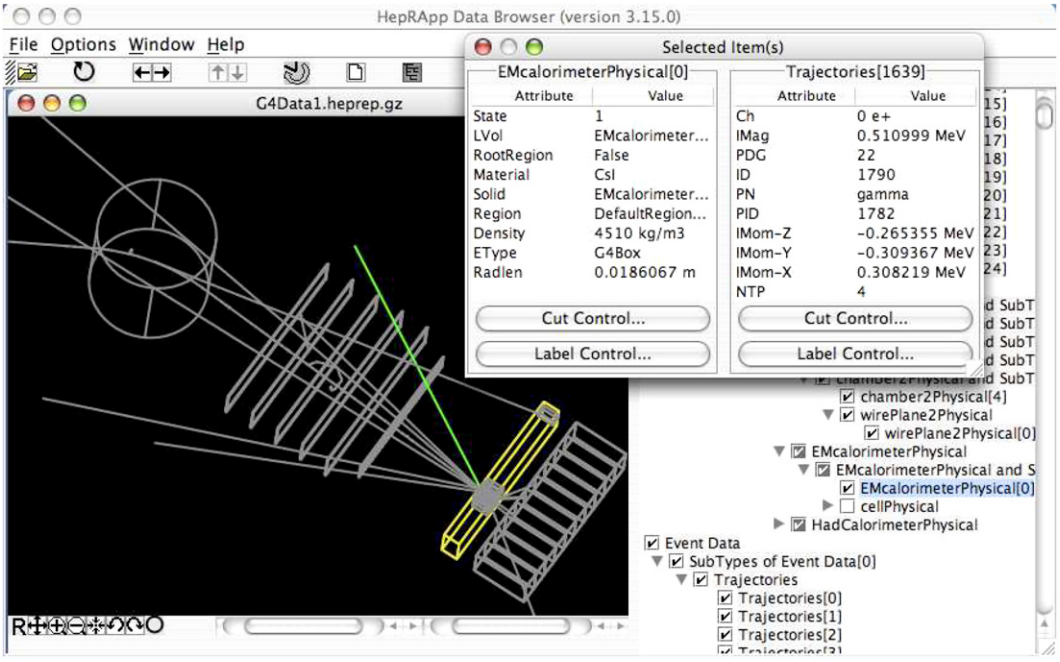

Fig. 7. Picked data from HepRApp with a HepRep file.

- displaying items as an interactive acyclic graph (tree), which allows to user to control:
  - the visibility of volumes;
  - the visibility of trajectories.

10.3. The OpenGL family

Most operating systems provide the basic but powerful graphics technology, OpenGL. If not, it can be obtained freely over the Internet. Consequently OpenGL has become the "workhorse" of the visualisation system. It has the advantage of being fast (most computers have OpenGL-based graphics accelerators) and showing the results of an action immediately on screen.

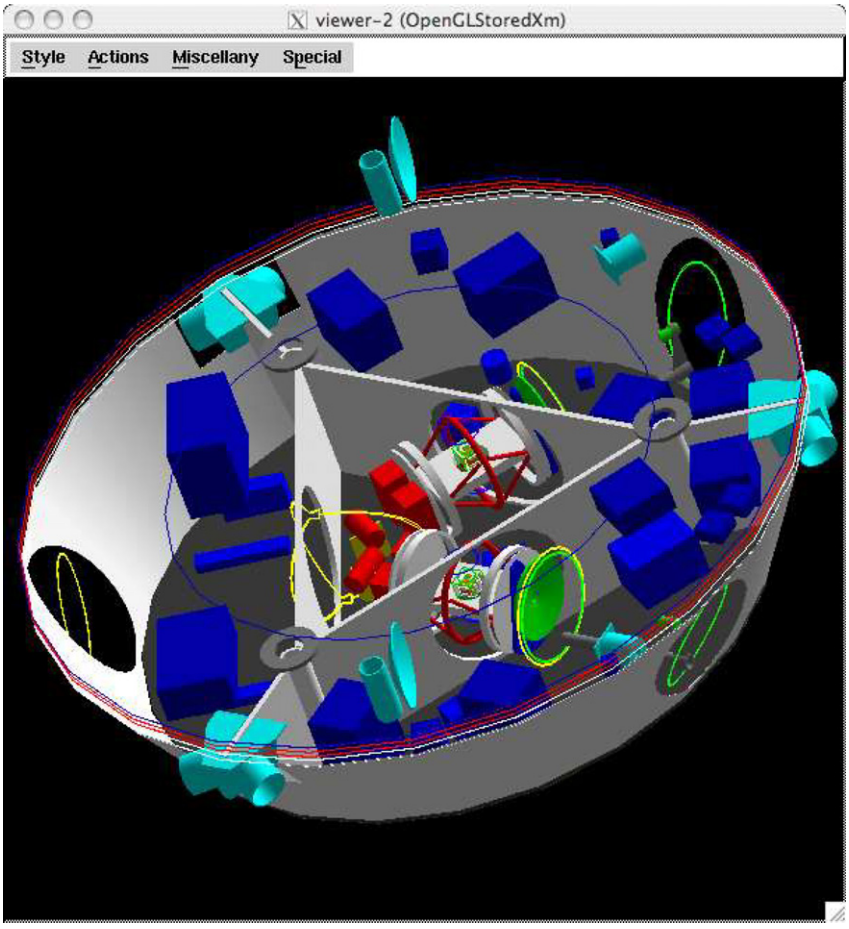

Fig. 8. OpenGL view of the LISA spacecraft.

For each window system—X, Xm and Win32—there are two graphics drivers: an "immediate" driver that simply draws to screen; and a "stored" driver that builds a graphical database (display lists) as it draws, thus allowing fast re-drawing on change of viewpoint, zoom factor, etc. The stored driver reverts to immediate mode in the case of memory overload.

Fig. 8 shows a window in the OpenGL stored Xm (OGLSXm) viewer. Xm is the common Motif window manager/toolkit for the X-Windows system and allows characteristic Motif-like interaction, as can be seen by the array of buttons on the top window bar. (Xm can also be obtained freely over the Internet as lesstif or OpenMotif.) Actions include rotation, zoom, panning, etc.

10.4. The Open Inventor family

Open Inventor is a C++ object-oriented wrapper for OpenGL. It too is freely available over the Internet. It supports extremely sophisticated interaction, as can be seen from Fig. 9. Geometry and track information is printed on picking. It also implements a geometry hierarchy representation whereby only the top-most visible volume is drawn; clicking on this volume makes it invisible and reveals its daughters. Thus, the user can reveal successive levels of detail interactively. This feature works best when drawing in surface style. The implementation of this feature is discussed in Appendix G.

It is also possible to write IV files for off-line browsing.

10.5. The Ray Tracer family

The Ray Tracer uses Geant4's own particle tracking system to trace optical rays. It is not only a test bed for the tracking system but also a way of getting photo-realistic renderings of the detector. By its nature, it is compute intensive. It differs from all the other graphics drivers in that it ignores the user-specified scene and simply renders the geometry "world". Thus, for example, it does not support trajectories. It does, however, respect the view parameters, and a viable procedure is to use OpenGL or Open Inventor to obtain a desired viewpoint, then:

```
/vis/open RayTracer
/vis/viewer/set/all <previous-viewer>
```

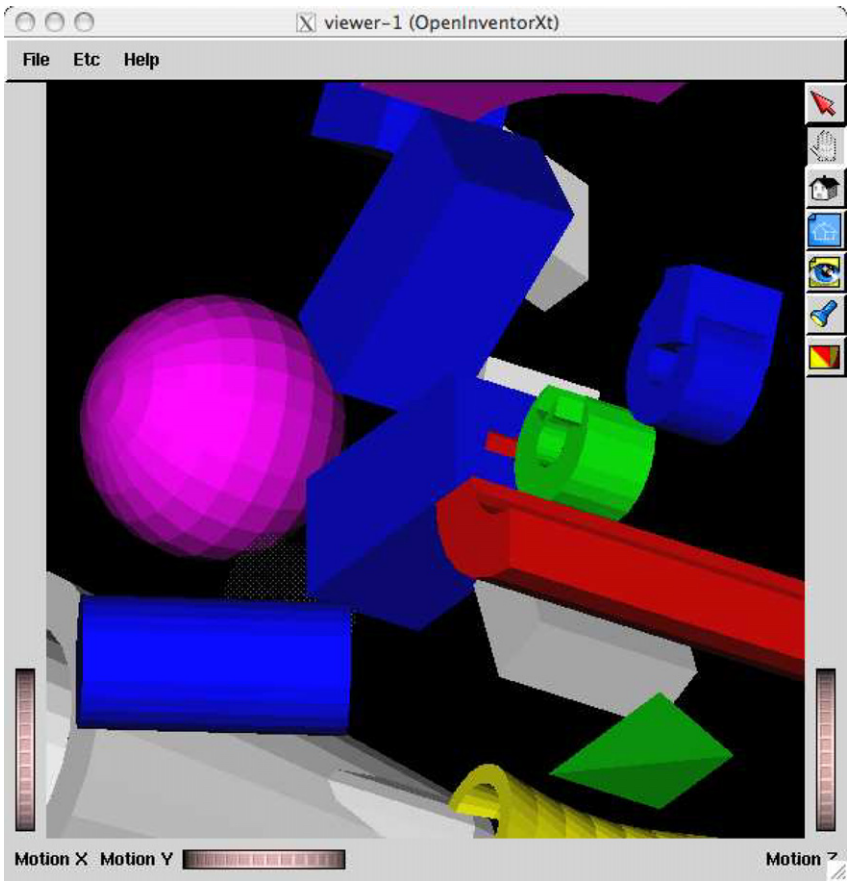

Fig. 9. Open Inventor view of a world of various shapes.

```
/vis/viewer/refresh
```

`RayTracerX` allows users of the X-Windows system to view the scene as it is progressively refined.

The Ray Tracer, via its own built-in commands, also allows the user to specify attenuation, distortion, shadows, etc. Fig. 10 shows some results.

10.6. The Tree family

This is a set of pseudo-graphics drivers that interpret the geometry hierarchy as an ASCII (text) representation. It is useful for seeing the hierarchical geometry structure and is a way of obtaining the names, volume, density and mass of the geometrical objects. A compound command `/vis/drawTree` is available for this. Fig. 11 shows an extract of a typical result.

10.7. The VRML family

Finally, the VRML family, like the DAWN family, writes to file or communicates via sockets to a VRML browser. Most Web browsers have VRML (Virtual Reality Modelling Language) plug-ins; free plug-ins or stand alone browsers may be obtained over the Internet.

11. Summary

We have described the GEANT4 Visualisation System, a sophisticated implementation of the GEANT4 graphics interface. Its power lies in being able to render GEANT4 objects—volumes, trajectories, etc.—in a number of different ways with a variety of graphics technologies. By selectively using the graphics driver's graphical database or, failing that, rebuilding from GEANT4's in-memory information, the GEANT4 Visualisation System produces uniform representations in all graphics technologies. Over and above that, most graphics drivers supply "value-added" features. Together they form a formidable set of tools to aid the application developer and the detector designer and enable the physicist to visualise the physics processes initiated by particles in matter.

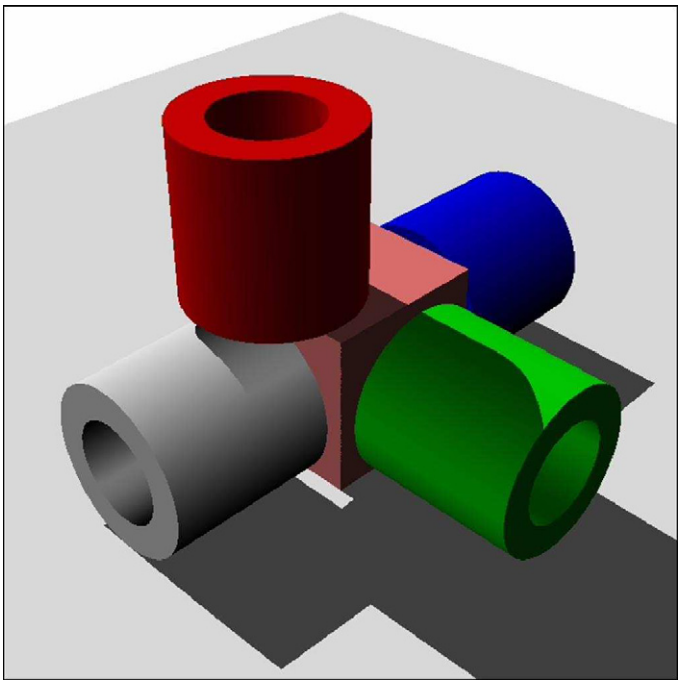

Fig. 10. A Ray Tracer rendering.

```
    "aTwistedBox-phys":0 / "aTwistedBox-log" /
    "aTwistedBox"(G4TwistedBox),
    360000 cm3, 1.782 mg/cm3, 360000 cm3, 641.52 g
Calculating mass(es)...
Overall volume of "expHall_P":0, is 8000 m3  and the
  daughter-included mass to unlimited depth
  is 76422.564 kg
```

Fig. 11. A fragment of the geometry tree shown by the ASCIITree driver for verbosity 15.

Being in C++ and written to abstract interfaces, the visualisation system is capable of being continually improved and augmented. The appendices give the detail needed by the toolkit developer who wishes to add a new graphics driver or model or filter, or incorporate GEANT4 into a specific graphics framework. Further information is available in the Toolkit Developers Guide [6].

Acknowledgements

Over the life of the GEANT4 Visualisation System there have been several significant contributions from developers who have moved on to other things. We would particularly like to mention and thank for their pioneering work: Joe Boudreau, University of Pittsburgh and Jeff Kallenbach, formerly of Fermi National Laboratory, for their contribution to the Open Inventor driver; Andy Walkden, as the major part of his PhD project at the University of Manchester, for the foundation work on the OpenGL drivers, particularly the Motif extensions, which introduce interactive control of the view. This work is supported in part by the U.S. Department of Energy under contract number DE-AC02-76SF00515.

Appendix A. Visualisation drivers

Table A.1 shows the currently available visualisation drivers. Depending on the graphics libraries available on a particular system, the installer must define some `G4VIS_BUILD...` C-pre-processor macros, such as `G4VIS_BUILD_OPENGLX_DRIVER`, so that the code for a particular driver is compiled (and inhibit compilation if the graphics libraries are not present). The make files distributed with the code interpret similarly named environment variables whose names conform to any defined by the "Environment ID" in Table A.1 and define the appropriate C-pre-processor macro during installation. A configuration script sets the environment variables in response to the installer's replies to a series of questions.

The visualisation system allows an application developer to instantiate the factory for any installed driver with code such as

```
visManager->RegisterGraphicsSystem(new G4OpenGLImmediateX);
```

Table A.1
Visualisation drivers. The "Environment ID" defines the environment variable for the standard make files, e.g., an environment ID OPENGLX means the environment variable G4VIS_BUILD_OPENGLX_DRIVER must be set for building libraries and G4VIS_USE_OPENGLX must be set when building the application

Class name	Environment ID	Nickname
G4ASCIITree	Needs none	ATree
G4DAWNFILE	"	DAWNFILE
G4HepRep	"	HepRepXML
G4HepRepFile	"	HepRepFile
G4RayTracer	"	RayTracer
G4VRML1File	"	VRML1FILE
G4VRML2File	"	VRML2FILE
G4FukuiRenderer	DAWN	DAWN
G4OpenGLImmediateX	OPENGLX	OGLIX
G4OpenGLStoredX	"	OGLSX
G4OpenGLImmediateWin32	OPENGLWIN32	OGLIWin32
G4OpenGLStoredWin32	"	OGLSWin32
G4OpenGLImmediateXm	OPENGLXM	OGLIXm
G4OpenGLStoredXm	"	OGLSXm
G4OpenInventorX	OIX	OIX
G4OpenInventorWin32	OIWIN32	OIWin32
G4RayTracerX	RAYTRACERX	RayTracerX
G4VRML1	VRML	VRML1
G4VRML2	"	VRML2

(Note that the Configure script in the standard code distribution sets these on response to simple questions.) The "Nickname" is the parameter used in the commands /vis/open or /vis/sceneHandler/create, e.g., /vis/open OGLIX.

A class G4VisExecutive is provided, as described above, that provides such code under control of G4VIS_USE... C-preprocessor macros. Again, the make files distributed with the code interpret similarly named environment variables and the configuration script sets all that may be set for a particular installation.

Once the factory has been instantiated, the drivers (i.e. scene handlers and viewers) can be instantiated with commands such as /vis/open or /vis/scenehandler/create giving the "Nickname" shown in the table as a parameter to the command.

Appendix B. The G4VGraphicsScene low-level interface (also known as the scene handler interface)

This is intended only for toolkit developers or users who provide their own visualisation system; it is used only by (a) the geometry category for describing shapes and (b) by models. A comment-stripped C++ class definition is shown in Fig. 3 in Section 5.

There are rules for invoking the functions of this interface:

- PreAddSolid, AddSolid and PostAddSolid must be invoked in that sequence. The transformation and visualisation attributes must be set by the call to PreAddSolid, which then apply to the solid passed to AddSolid (only one AddSolid is allowed). A possible default implementation is to request the solid to provide a G4Polyhedron or similar primitive—see, for example, G4VSceneHandler in the Visualisation Category.
- A sequence of calls to AddPrimitive must be sandwiched between calls to BeginPrimitives and EndPrimitives or BeginPrimitives2D and EndPrimitives2D. A sequence is any number of calls that have the same transformation.
- For Begin/EndPrimitives2D, the x-, y-coordinates of the primitives passed to AddPrimitive are interpreted as screen coordinates, $-1 < x, y < 1$. The z-coordinate is ignored.

Note that objectTransformation is the transformation which transforms the coordinates of the object into its location in the world (top-level) coordinate system. It is not a transformation of coordinate systems (which is the inverse). Also note that visAttribs must be supplied for solids through PreAddSolid, whereas the attributes for basic graphical objects are to be obtained through their GetVisAttributes method.

In the GEANT4 Visualisation System, G4VGraphicsScene is extended by the class G4VSceneHandler. Concrete instances of the latter class are called *scene handlers* in the parlance of the GEANT4 Visualisation System. Each scene handler is specific to the underlying graphics technology. Any implementation of G4VSceneHandler may assume the above rules have been adhered to by the users. How this is exploited, and additional features of G4VSceneHandler, are discussed in Section D.3.

Appendix C. Visualisation attributes

The class G4VisAttributes is introduced in Section 4. A list of attributes is shown in Table 1. Note that the visualisation attributes include (and should not be confused with) pointers to general attributes, G4AttDef and G4AttValue, that may be used for additional information about an object—see Appendix J.

The following considerations need to be taken into account by the designer/implementer of a scene handler.

All drawable objects are required to have a method:

```
const G4VisAttributes* GetVisAttributes() const;
```

A drawable object might be:

- a "visible" (i.e., inheriting G4Visible), such as a polyhedron, polyline, circle, etc. (note that text is a slightly special case—see below), or
- a solid whose attributes are held in its logical volume.

C.1. Finding the applicable attributes

This is an issue for all scene handlers. The scene handler is where the colour, style, auxiliary edge visibility, marker size, etc., of individual drawable objects are needed. The attributes might have been set in PreAddSolid, or might be attached the drawable object, or might not be set at all. In the last case, the pointer will be zero and the scene handler must obtain the default attributes from the viewer. A utility function G4VViewer::GetApplicableVisAttributes is provided, so, when dealing with primitives, the scene handler should obtain the attributes as follows:

```
const G4VisAttributes* pVisAtts =
  fpViewer->GetApplicableVisAttributes(polyline.GetVisAttributes());
```

This is taken into account in the utility function that provides colour:

```
const G4Colour& colour = GetTextColour(visible);
```

where visible is the appropriate drawable object—polyhedron, circle, square, etc.

Once the applicable attributes have been found, G4VSceneHandler provides utility functions that obtain some of the less straightforward quantities needed for drawing, for example, where parameters may be overridden (forced) or modified by the visualisation attributes:

```
G4double GetLineWidth(const G4VisAttributes*);
G4ViewParameters::DrawingStyle GetDrawingStyle (const G4VisAttributes*);
G4bool GetAuxEdgeVisible (const G4VisAttributes*);
G4int GetNoOfSides(const G4VisAttributes*);
```

Other utility functions interrogate a marker and return the appropriate quantity and MarkerSizeType, based on the applicable visualisation attributes:

```
G4double GetMarkerSize (const G4VMarker&, MarkerSizeType&);
G4double GetMarkerDiameter (const G4VMarker&, MarkerSizeType&);
G4double GetMarkerRadius (const G4VMarker&, MarkerSizeType&);
```

Their use is illustrated in some of the following sections.

C.2. Text

Text is a special case because it has its own default attributes:

```
const G4VisAttributes* pVisAtts = text.GetVisAttributes();
if (!pVisAtts)
  pVisAtts = fpViewer->GetViewParameters().GetDefaultTextVisAttributes();
```

and as above, there is a utility function for colour that takes this into account:

```
const G4Colour& colour = GetTextColour(text);
```

C.3. Solids

For specific solids, the G4PhysicalVolumeModel that provides the solids also provides, via PreAddSolid, a pointer to its attributes. If the attributes pointer in the logical volume is zero, it provides a pointer to the default attributes in the model, which in turn is (in the GEANT4 Visualisation System) provided by the viewer's attributes (see G4VSceneHandler::CreateModelingParameters). So the attributes pointer is guaranteed to be pertinent.

If the concrete driver does not implement AddSolid for any particular solid, the base class converts it to primitives (usually a G4Polyhedron) and again, the attributes pointer is guaranteed.

C.4. Drawing style

The drawing style is normally determined by the view parameters but for individual drawable objects it may be overridden by the forced drawing style flags in the attributes. A utility function is provided that accounts for this:

```
G4ViewParameters::DrawingStyle drawing_style = GetDrawingStyle(pVisAtts);
```

C.5. Auxiliary edges

Similarly, the visibility of auxiliary/soft edges is normally determined by the view parameters but may be overridden by the forced auxiliary edge visible flag in the attributes. Again, a utility function is provided:

```
G4bool isAuxEdgeVisible = GetAuxEdgeVisible (pVisAtts);
```

C.6. Polygon approximation

For visualisation of volumes with edges that are (a part of) a circle, the circle is represented by an N-sided polygon. The default is 24 sides or segments. The user may change this for all volumes in a particular viewer at run time with /vis/viewer/set/lineSegmentsPerCircle; alternatively it can be overridden for a particular volume by the forced line segments per circle parameter in the attributes. A utility is provided that respects this:

```
G4int lineSegmentsperCircle = GetNoOfSides (pVisAtts);
```

C.7. Marker size

These are an extra property of markers, i.e. objects that inherit G4VMarker (circles, squares, text, etc.), over and above visualisation attributes. However, the algorithm for the actual size is quite complicated and a utility function is provided:

```
MarkerSizeType sizeType;
G4double size = GetMarkerSize(text, sizeType);
```

sizeType is world or screen, signifying that the size is in world coordinates or screen coordinates, respectively.

Appendix D. Creating a new graphics driver

To create a new graphics driver for GEANT4, it is necessary to implement a new set of three classes derived from the three base classes, G4VGraphicsSystem, G4VSceneHandler and G4VViewer.

D.1. A useful place to start

A skeleton set of classes is included in the code distribution in the visualisation category under subdirectory visualisation/XXX (but they are not default-registered graphics drivers[5]). There are several sets of classes, described in more detail below. A recommended approach is to copy the files that best match your graphics technology to a new subdirectory with a name that suits your graphics technology. Then

(1) Change the name of the files (change the code—XXX or XXXFile, etc., as chosen—to something that suits your graphics technology).

[5] To do this, simply instantiate and register, for example: visManager->RegisterGraphicsSystem(new G4XXX) before visManager->Initialise().

(2) Change XXX similarly in all files.
(3) Change XXX similarly in name := G4XXX in GNUmakefile.
(4) Add your new subdirectory to SUBDIRS and SUBLIBS in visualisation/GNUmakefile.
(5) Look at the code and use it to build your visualisation driver. You might also find it useful to look at ASCIITree (and VTree) as an example of a minimal graphics driver. Look at FukuiRenderer as an example of a driver which implements AddSolid methods for some solids. Look at OpenGL as an example of a driver which implements a graphical database (display lists) and the machinery to decide when to rebuild. (OpenGL is complicated by the proliferation of combinations of the use or not of display lists for three window systems, X-windows, X with motif (interactive), Microsoft Windows (Win32), a total of six combinations, and much use is made of inheritance to avoid code duplication.)
(6) If it requires external libraries, introduce two new environment variables G4VIS_BUILD_XXX_DRIVER and G4VIS_USE_XXX (where XXX is your choice as above) and make the modifications to:

- source/visualization/management/include/G4VisExecutive.icc,
- config/G4VIS_BUILD.gmk,
- config/G4VIS_USE.gmk.

You may use the following templates in the XXX sub-category to help you get started writing a graphics driver. (The word "template" is used in the ordinary sense of the word; they are not C++ templates.)

- G4XXX, G4XXXSceneHandler, G4XXXViewer. Templates for the simplest possible graphics driver. These would be suitable for an "immediate" driver, i.e., one which renders each object immediately to a screen. Of course, if the view needs re-drawing, as, for example, after a change of viewpoint, the viewer requests a re-issue of drawn objects.
- G4XXXFile, G4XXXFileSceneHandler, G4XXXFileViewer. Templates for a file-writing graphics driver. The particular features are: delayed opening of the file on receipt of the first item; rewinding file on ClearView (to simulate the clearing of views and prevent the duplication of material in the file); closing of the file on ShowView, which may also trigger the launch of a browser. There are various degrees of sophistication in, for example, the allocation of filenames—see FukuiRenderer or HepRepFile.
  These templates also show the use of a specific AddSolid function whereby the specific parameters, for example, the dimensions of a G4Box, can be accessed.
- G4XXXStored, G4XXXStoredSceneHandler, G4XXXStoredViewer. Templates for a graphics driver with a store/database. The advantage of a store is that the view can be refreshed, for example, from a different viewpoint, without a need to recompute. It is up to the viewer to decide when a re-computation is necessary. They also show how to distinguish between permanent and transient objects—see also Appendix F.
- G4XXXSG, G4XXXSGSceneHandler, G4XXXSGViewer. Templates for a sophisticated graphics driver with a scene graph. The scene graph, following Open Inventor parlance, is a tree of objects that dictates the order in which the objects are rendered. It obviously lends itself to the rendering of the GEANT4 geometry hierarchy. For example, the Open Inventor driver draws only the top level volumes unless made invisible by picking. Thus the user can unwrap a branch of the geometry level by level. This has performance benefits and gives the user significant and useful control over the view. These classes show how to make a scene graph of *drawn* volumes, i.e. the set of volumes that have not been culled. (Normally, volumes marked invisible are culled, i.e. not drawn. Also, the user may wish to limit the number of drawn volumes for performance reasons.) The drivers also have to process non-geometry items and distinguish between transient and permanent objects as above.

D.2. The G4VGraphicsSystem base class

This is a factory for scene handlers and viewers. Its constructor should invoke the full base-class constructor that specifies name, nickname, description and "functionality". The last is intended to indicate the type of graphics driver and is defined by the following:

```
enum Functionality {
  noFunctionality,
  nonEuclidian,       // e.g., tree representation of geometry hierarchy.
  twoD,               // Simple 2D, e.g., X (no stored structures).
  twoDStore,          // 2D with stored structures.
  threeD,             // Passive 3D (with stored structures).
  threeDInteractive,  // 3D with "pick" functionality.
  virtualReality      // Virtual Reality functionality.
};
```

A typical constructor is:

```
G4OpenGLImmediateX::G4OpenGLImmediateX():
  G4VGraphicsSystem ("OpenGLImmediateX",
                     "OGLIX",
    "    Dumb single buffered X Window, No Graphics Database."
    "\n Advantages: does not gobble server memory."
    "\n good for drawing steps and hits."
    "\n Disadvantages: needs G4 kernel for re-Draw."
    "\n cannot take advantage of graphics accelerators.",
                     G4VGraphicsSystem::threeD) {}
```

It is required to implement:

```
virtual G4VSceneHandler* CreateSceneHandler (const G4String& name) = 0;
virtual G4VViewer* CreateViewer (G4VSceneHandler&, const G4String& name) = 0;
```

and typical implementations are:

```
G4VSceneHandler* G4OpenGLImmediateX::CreateSceneHandler(const G4String& name) {
  G4VSceneHandler* pScene = new G4OpenGLImmediateSceneHandler (*this, name);
  return    pScene;
}

G4VViewer* G4OpenGLImmediateX::CreateViewer
(G4VSceneHandler& scene, const G4String& name) {
  G4VViewer* pView =
    new G4OpenGLImmediateXViewer((G4OpenGLImmediateSceneHandler&)scene, name);
  if (pView) {
    if (pView->GetViewId() < 0) {
      G4cerr << "G4OpenGLImmediateX::CreateViewer: error flagged by negative"
        " view id in G4OpenGLImmediateXViewer creation."
        "\n Destroying view and returning null pointer."
          << G4endl;
      delete pView;
      pView = 0;
    }
  }
  else {
    G4cerr << "G4OpenGLImmediateX::CreateViewer: null pointer on"
      " new G4OpenGLImmediateXViewer." << G4endl;
  }
  return pView;
}
```

Notice the use of `GetViewId` to signal an error in the creation of the viewer. Also, if the viewer needs access to its specific scene handler, this can be done with the cast in the viewer construction. (Perhaps a more modern approach would be to use a dynamic cast in the viewer when required.)

D.3. The `G4VSceneHandler` base class

This conforms to and extends the `G4VGraphicsScene` low-level interface—see Figs. 3 and D.1 in Sections 5 and Appendix B. The implementer may assume that the rules of invocation described in Section Appendix B have been adhered to. In summary, these are:

- `PreAddSolid`, `AddSolid` and `PostAddSolid` must be invoked in that sequence.
- A sequence of calls to `AddPrimitive` must be sandwiched between calls to `BeginPrimitives` and `EndPrimitives` or `BeginPrimitives2D` and `EndPrimitives2D`.

Certain pairs of virtual methods must be *extended* rather than over-ridden. This simply means that the polymorphic method of the concrete scene handler must explicitly invoke the base class method. For example:

```
class G4VSceneHandler: public G4VGraphicsScene {
  ...methods of G4VGraphicsScene plus...
  virtual void BeginModeling ();
  virtual void EndModeling ();
  virtual void ClearStore ();
  virtual void ClearTransientStore ();
  const G4Colour& GetTextColour (const G4Text&);
  const G4Colour& GetTextColor  (const G4Text&);
  G4double GetLineWidth(const G4VisAttributes*);
  G4ViewParameters::DrawingStyle GetDrawingStyle (const G4VisAttributes*);
  G4bool GetAuxEdgeVisible (const G4VisAttributes*);
  G4int GetNoOfSides(const G4VisAttributes*);
  G4double GetMarkerSize (const G4VMarker&, MarkerSizeType&);
  G4double GetMarkerDiameter (const G4VMarker&, MarkerSizeType&);
  G4double GetMarkerRadius (const G4VMarker&, MarkerSizeType&);
  G4ModelingParameters* CreateModelingParameters ();
  ...plus some access functions.
};
```

Fig. D.1. The `G4VSceneHandler` extended base class.

```
void MyXXXSceneHandler::BeginModeling () {
  G4VSceneHandler::BeginModeling ();
  ...
}
void MyXXXSceneHandler::EndModeling () {
  ...
  G4VSceneHandler::EndModeling ();
}
```

Here is a complete list of methods with a similar requirement:

```
BeginModeling        EndModeling
PreAddSolid          PostAddSolid
BeginPrimitives      EndPrimitives
BeginPrimitives2D    EndPrimitives2D
ClearStore
ClearTransientStore
```

The pair of additional functions `Begin/EndModeling` bracket the processing of the run-duration part of the scene, which allows the scene handler to initialise and consolidate, respectively, the processing of permanent objects. The scene handler base class maintains a flag `fReadyForTransients`, which is `false` only within the `Begin/EndModeling` scope.

As stated in Section 7.1—and worth repeating here—is that if the scene handler needs any information from a model, for example, to obtain textual tags or build its own scene graph, it is necessary to make a dynamic cast:

```
G4PhysicalVolumeModel* pPVModel =
  dynamic_cast<G4PhysicalVolumeModel*>(fpModel);
if (pPVModel) {
  tag = pPVModel->GetCurrentPV()->GetName();
  ...
} else {
  // Not from a G4PhysicalVolumeModel.
...
```

Note that the above requirement to make a dynamic_cast deals with the possibility that the current object is *not* generated by a model—for example, it may come direct from user code—in which case `fpModel` itself will be zero.

Appendix G discusses the building of drawn scene graphs. Note that it is always possible that the solid does *not* come from a `G4PhysicalVolumeModel` (perhaps it is a transient volume representing a hit) in which case it must be treated differently, with no place in the geometry hierarchy.

```
class G4VViewer {
public:
  G4VViewer (G4VSceneHandler&, G4int id, const G4String& name = "");
  virtual  G4VViewer();
  virtual void Initialise();
  virtual void SetView() = 0;
  virtual void ClearView() = 0;
  virtual void DrawView() = 0;
  virtual void ShowView();
  virtual void FinishView();
  const G4VisAttributes*  GetApplicableVisAttributes
    (const G4VisAttributes*) const;
  void SetNeedKernelVisit (G4bool need);
  void NeedKernelVisit();
  void ProcessView();
  ...
};
```

Fig. D.2. The essential `G4VViewer` interface.

The pair `ClearStore` and `ClearTransientStore` were designed for scene handlers that have their own graphical database/scene graph/store. Scene handlers that write files must emulate the clearing of the store by rewinding (or rewriting) the output file.

The tricky issue of building a database that can handle all possible function invocation sequences is discussed in Appendix E. For convenience, the scene handler base class maintains a flag `fProcessingSolid`, which is `true` only within the `Pre/PostAddSolid` scope. At the same time, the permanent and transient parts of the database must be distinguished, so the `ClearTransientStore` clears only the transient part. Transients are described in Appendix F. The building of a scene graph is discussed in Appendix G. These issues are reflected in design of the templates `XXXStored` and `XXXSG` described in Section D.1.

D.4. The `G4VViewer` base class

Fig. D.2 shows that a minimum of three functions are required to be implemented by any concrete viewer—`SetView` (to interpret the view parameters, Fig. D.3), `ClearView` and `DrawView`. Appendix E describes in detail the circumstances under which these functions are called.

Other functions also play important roles:

- `Initialise` Called immediately after construction. Useful for viewers that need to await complete construction before certain operations can be performed. Currently only used by OpenGL (see comments in `G4VViewer.hh`).
- `ShowView` For viewers that need to trigger post-processing of the drawn information, for example, to perform hidden line removal or close a file. It is called when the user issues `/vis/viewer/update` or at the end of event or run, depending of scene actions.
- `FinishView` This is called after processing the run-duration models and before end-of-event models (if requested). It is intended for flushing buffers (to ensure all drawn material is written to screen) and/or swapping front and back buffers (in double buffer systems). It is also called when the user issues `/vis/viewer/clear`. Otherwise, it may be used internally by the viewer.
- `SetNeedKernelVisit` and `NeedKernelVisit` Alternative ways of recording the need for a kernel visit, i.e. when the scene needs to be processed again. Viewers of graphics drivers that render the scene to a graphical database need not re-visit the kernel for simple operations such as change of viewpoint. This should be determined in `DrawView`. Two examples are shown in Figs. D.4 and D.5.
- `ProcessView` This is the first function encountered. If a kernel visit is requested it calls the scene handler's `ProcessScene`.

Appendix E. Important command actions

To help understand how the GEANT4 Visualisation System works, here are a few important function invocation sequences that follow user commands. What happens in `DrawView`, `ProcessView` and `ProcessScene` will be discussed in detail below. For an explanation of the commands themselves, see the command guidance or the Control section of the Application Developers Guide [2]. For a fuller explanation of the functions, see appropriate base class header files or Software Reference Manual [5].

```
DrawingStyle fDrawingStyle;    // Drawing style.
G4bool       fAuxEdgeVisible;  // Auxiliary edge visibility.
RepStyle     fRepStyle;        // Representation style.
G4bool       fCulling;         // Culling requested.
G4bool       fCullInvisible;   // Cull (don't Draw) invisible objects.
G4bool       fDensityCulling;  // Density culling requested.  If so...
G4double     fVisibleDensity;  // ...density lower than this not drawn.
G4bool       fCullCovered;     // Cull daughters covered by opaque mothers.
G4bool       fSection;         // Section drawing requested (DCUT in GEANT3).
G4Plane3D    fSectionPlane;    // Cut plane for section drawing (DCUT).
CutawayMode  fCutawayMode;     // Cutaway mode.
G4Planes     fCutawayPlanes;   // Set of planes used for cutaway.
G4bool       fExplode;         // Explode flag.
G4double     fExplodeFactor;
G4int        fNoOfSides;       // ...if polygon approximates circle.
G4Vector3D   fViewpointDirection;
G4Vector3D   fUpVector;        // Up vector.  (Warning: MUST NOT be parallel
                               // to fViewpointDirection!)
G4double     fFieldHalfAngle;  // Radius / camera distance, 0 for parallel.
G4double     fZoomFactor;      // Magnification relative to Standard View.
G4Vector3D   fScaleFactor;     // (Non-uniform) scale/magnification factor.
G4Point3D    fCurrentTargetPoint;  // Relative to standard target point.
G4double     fDolly;           // Distance towards current target point.
G4bool       fLightsMoveWithCamera;
G4Vector3D   fRelativeLightpointDirection;
// i.e., rel. to object or camera according to G4bool fLightsMoveWithCamera.
G4Vector3D   fActualLightpointDirection;
G4VisAttributes fDefaultVisAttributes;
G4VisAttributes fDefaultTextVisAttributes;
G4VMarker    fDefaultMarker;
G4double     fGlobalMarkerScale;
G4bool       fMarkerNotHidden;
// True if transients are to be drawn and not hidden by
// hidden-line-hidden-surface removal algorithms, e.g., z-buffer
// testing; false if they are to be hidden-line-hidden-surface
// removed.
G4int        fWindowSizeHintX; // Size hints for pixel-based window systems.
G4int        fWindowSizeHintY;
G4String     fXGeometryString; // If non-null, geometry string for X Windows.
G4bool       fAutoRefresh;     // ...after change of view parameters.
G4Colour     fBackgroundColour;
```

Fig. D.3. View parameters (from `G4ViewParameters`).

```
void G4HepRepFileViewer::DrawView() {
  NeedKernelVisit();
  ProcessView();
}
```

Fig. D.4. `DrawView` for a simple viewer.

- `/vis/viewer/clear`

```
viewer->ClearView();   // Clears buffer or rewinds file.
viewer->FinishView();  // Swaps buffer (double buffer systems).
```

- `/vis/viewer/flush`

```
/vis/viewer/refresh
/vis/viewer/update
```

```
void G4OpenGLStoredXViewer::DrawView() {
  if (!fNeedKernelVisit) KernelVisitDecision();
  // Keep decision (ProcessView resets)...
  G4bool kernelVisitWasNeeded = fNeedKernelVisit;
  ProcessView();
  if (!kernelVisitWasNeeded) {
    DrawDisplayLists();
    FinishView();
  }
}
void G4OpenGLStoredViewer::KernelVisitDecision() {
  if (CompareForKernelVisit(fLastVP)) NeedKernelVisit();
  fLastVP = fVP;
}
G4bool G4OpenGLStoredViewer::CompareForKernelVisit
(G4ViewParameters& lastVP) {
  if ((lastVP.GetDrawingStyle() != fVP.GetDrawingStyle()) ||
      ...) return true;
  ...
  return false;
}
```

Fig. D.5. `DrawView` for a graphics driver with database.

- `/vis/viewer/rebuild`

  ```
  viewer->SetNeedKernelVisit(true);
  ```

- `/vis/viewer/refresh`. If the view is "auto-refresh",[6] this command is also invoked after `/vis/viewer/create`, `/vis/viewer/rebuild` or a change of view parameters (`/vis/viewer/set/...`, etc.).

  ```
  viewer->SetView();    // Sets camera position, etc.
  viewer->ClearView();  // Clears buffer or rewinds file.
  viewer->DrawView();   // Draws to screen or writes to file/socket.
  ```

- `/vis/viewer/update`

  ```
  viewer->ShowView();   // Activates interactive windows or closes file
                        // and/or triggers post-processing.
  ```

- `/vis/scene/notifyHandlers`. For each viewer of the current scene, the equivalent of

  `/vis/viewer/refresh`

  If "flush" is specified on the command line, the equivalent of

  `/vis/viewer/update`

  `/vis/scene/notifyHandlers` is also invoked after a change of scene (`/vis/scene/add/...`, etc.).

E.1. What happens in `DrawView`?

This depends on the viewer. Those with their own graphical database, for example, OpenGL's display lists or Open Inventor's scene graph, do not need to re-traverse the scene unless there has been a significant change of view parameters. For example, a mere change of viewpoint requires only a change of model-view matrix whilst a change of rendering mode from wireframe to surface might require a rebuild of the graphical database. A rebuild of the run-duration (persistent) objects in the scene is called a kernel visit; the viewer prints "Traversing scene data...".

[6] The auto-refresh flag is a member of `G4ViewParameters`. Its default value is `false`. However, the OpenGL and Open Inventor viewers set it to `true` in their constructors. It may also be set/unset by the command `/vis/viewer/set/autoRefresh`.

Note that end-of-event (transient) objects are only rebuilt at the end of an event or run, under control of the visualisation manager. Smart scene handlers keep them in separate display lists so that they can be rebuilt separately from the run-duration objects—see Appendix F.

- **Integrated viewers with no graphical database.** For example, `G4OpenGLImmediateXViewer::DrawView()`.

```
NeedKernelVisit();  // Always need to visit G4 kernel.
ProcessView();
FinishView();
```

- **Integrated viewers with graphical database.** For example, `G4OpenGLStoredXViewer::DrawView()`.

```
//See if things have changed from last time and remake if necessary...
// The fNeedKernelVisit flag might have been set by the user in
// /vis/viewer/rebuild, but if not, make decision and set flag only
// if necessary...
if (!fNeedKernelVisit) KernelVisitDecision ();
// KernelVisitDecision is a private function that sets fNeedKernelVisit.
// Keep copy of fNeedKernelVisit - ProcessView resets.
G4bool kernelVisitWasNeeded = fNeedKernelVisit;
ProcessView ();
// If kernel visit was needed, drawing and FinishView will already
// have been done, so...
if (!kernelVisitWasNeeded) {
  DrawDisplayLists ();
  FinishView ();  // Swaps buffers.
}
```

- **File-writing viewers.** For example, `G4DAWNFILEViewer::DrawView()`.

```
NeedKernelVisit();
ProcessView();
```

Note that viewers needing to invoke `FinishView` must do it in `DrawView`.

E.2. What happens in `ProcessView`?

This is invoked by `DrawView` if appropriate conditions are satisfied, as described in the previous section.

```
void G4VViewer::ProcessView() {
  // If ClearStore has been requested, e.g., if the scene has changed,
  // of if the concrete viewer has decided that it necessary to visit
  // the kernel, perhaps because the view parameters have changed
  // drastically (this should be done in the concrete viewer's
  // DrawView)...
  if (fNeedKernelVisit) {
    fSceneHandler.ProcessScene(*this);
    fNeedKernelVisit = false;
  }
}
```

E.3. What happens in `ProcessScene`?

`ProcessScene` causes a traversal of the scene. For drivers with graphical databases, it causes a rebuild (`ClearStore`). In outline:

```
fReadyForTransients = false;
BeginModeling();
for each run-duration model...
  pModel -> DescribeYourselfTo(*this);
```

```
void G4HepRepFileSceneHandler::AddSolid(const G4Box& box) {
        ...
        G4double dx = box.GetXHalfLength();
        G4double dy = box.GetYHalfLength();
        G4double dz = box.GetZHalfLength();
        G4Point3D vertex1(G4Point3D( dx, dy,-dz));
        ...
        vertex1 = (*fpObjectTransformation) * vertex1;
        ...
        hepRepXMLWriter->addPoint(vertex1.x(), vertex1.y(), vertex1.z());
        ...
```

Fig. E.1. `DrawView` for a graphics driver with database.

```
EndModeling();
fReadyForTransients = true;
for each event available, for each end-of-event model...
  pModel -> DescribeYourselfTo(*this);
```

(A second pass is made on request—see `G4VSceneHandler::ProcessScene`.) The use of `fReadyForTransients` is described in Appendix F.

What happens then depends on the type of model:

- `G4AxesModel` `G4AxesModel::DescribeYourselfTo` simply calls sceneHandler.AddPrimitive methods directly.

```
sceneHandler.BeginPrimitives();
sceneHandler.AddPrimitive(x_axis);  // etc.
sceneHandler.EndPrimitives();
```

Most other models are like this, except for the following…

- `G4PhysicalVolumeModel`. The geometry is descended recursively, culling policy is enacted, and for each accepted (and possibly, clipped[7]) solid:

```
sceneHandler.PreAddSolid(theAT, *pVisAttribs);
pSol->DescribeYourselfTo(sceneHandler);
// For example, if pSol points to a G4Box...
|-->G4Box::DescribeYourselfTo(G4VGraphicsScene& scene){
     scene.AddSolid(*this);
    }
sceneHandler.PostAddSolid();
```

The scene handler may implement the virtual function `AddSolid(const G4Box&)` (an example is shown in Fig. E.1), or inherit:

```
void G4VSceneHandler::AddSolid(const G4Box& box) {
  RequestPrimitives(box);
}
```

`RequestPrimitives` converts the solid into primitives (`G4Polyhedron`) and invokes `AddPrimitive`:

```
BeginPrimitives(*fpObjectTransformation);
pPolyhedron = solid.GetPolyhedron();
AddPrimitive(*pPolyhedron);
EndPrimitives();
```

The resulting default sequence for a `G4PhysicalVolumeModel` is shown in Fig. E.2.

Note the sequence of calls at the core:

[7] Clipped solids are converted into `G4Polyhedron` objects for Boolean operations and dispatched to the scene handler as such.

```
DrawView();
|-->ProcessView();
    |-->ProcessScene();
        |-->BeginModeling();
        |-->pModel -> DescribeYourselfTo(*this);
        |   |-->sceneHandler.PreAddSolid(theAT, *pVisAttribs);
        |   |-->pSol->DescribeYourselfTo(sceneHandler);
        |   |   |-->sceneHandler.AddSolid(*this);
        |   |       |-->RequestPrimitives(solid);
        |   |           |-->BeginPrimitives (*fpObjectTransformation);
        |   |           |-->pPolyhedron = solid.GetPolyhedron();
        |   |           |-->AddPrimitive(*pPolyhedron);
        |   |           |-->EndPrimitives();
        |   |-->sceneHandler.PostAddSolid();
        |-->EndModeling();
```

Fig. E.2. The default sequence for a `G4PhysicalVolumeModel`.

```
sceneHandler.PreAddSolid(theAT, *pVisAttribs);
pSol->DescribeYourselfTo(sceneHandler);
|-->sceneHandler.AddSolid(*this);
   |-->RequestPrimitives(solid);
       |-->BeginPrimitives (*fpObjectTransformation);
       |-->pPolyhedron = solid.GetPolyhedron();
       |-->AddPrimitive(*pPolyhedron);
       |-->EndPrimitives();
sceneHandler.PostAddSolid();
```

is reduced to

```
sceneHandler.PreAddSolid(theAT, *pVisAttribs);
pSol->DescribeYourselfTo(sceneHandler);
|-->sceneHandler.AddSolid(*this);
sceneHandler.PostAddSolid();
```

if the scene handler implements its own `AddSolid`. Moreover, the sequence

```
BeginPrimitives (*fpObjectTransformation);
AddPrimitive(*pPolyhedron);
EndPrimitives();
```

can be invoked without a prior `PreAddSolid`, etc. The flag `fProcessingSolid` will be false for the last case. The possibility of any or all of these three scenarios occurring, for both permanent and transient objects, affects the implementation of a scene handler if there is any attempt to build a graphical database. This is reflected in the templates `XXXStored` and `XXXSG` described in Section D.1. Transients are discussed in Appendix F and the building of a scene graph is discussed in Appendix G.

- `G4TrajectoriesModel` At end of event, the trajectory container is unpacked and, for each trajectory, `sceneHandler.AddCompound` called. The scene handler may implement this virtual function or inherit:

```
void G4VSceneHandler::AddCompound (const G4VTrajectory& traj) {
  traj.DrawTrajectory(((G4TrajectoriesModel*)fpModel)->GetDrawingMode());
}
```

Similarly, the user may implement `DrawTrajectory` or inherit:

```
void G4VTrajectory::DrawTrajectory(G4int i_mode) const {
  G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
  if (0 != pVVisManager) {
    pVVisManager->DispatchToModel(*this, i_mode);
  }
}
```

```
DrawView();
|-->ProcessView();
    |-->ProcessScene();
        |-->BeginModeling();
        |-->pModel -> DescribeYourselfTo(*this);
        |   |-->AddCompound(trajectory);
        |       |-->trajectory.DrawTrajectory(...);
        |           |-->DispatchToModel(...);
        |               |-->model->Draw(...);
        |                   |-->G4VisManager::Draw(...);
        |                       |-->BeginPrimitives(objectTransform);
        |                       |-->AddPrimitive(...);
        |                       |-->EndPrimitives();
        |-->EndModeling();
```

Fig. E.3. The default sequence for a `G4TrajectoriesModel`.

Thence, the `Draw` method of the current trajectory model is invoked (see Appendix H for details on trajectory models), which in turn, invokes `Draw` methods of the visualisation manager. The resulting default sequence for a `G4TrajectoriesModel` is shown in Fig. E.3.

- **End-of-event models.** End-of-event models include: `G4TrajectoriesModel`, `G4HitsModel` and `G4CallbackModel` that represents the current event ID. These are normally processed at the end of event in `G4VisManager::EndOfEvent`. However, they are also processed in `ProcessScene` if the run manager has kept an event. (Events may be kept by the user; alternatively, the visualisation manager asks the run manager to keep the last event of the run or, if `/vis/scene/endOfEventAction accumulate <N>`, N events.)

Appendix F. Transient and permanent drawing

Any visualisable object not defined in the run-duration part of a scene is treated as "transient". This includes trajectories, hits and anything drawn by the user through the `G4VVisManager` user interface (unless as part of a run-duration model implementation). A flag, `fReadyForTransients`, is maintained by the scene handler. In fact, its normal state is `true`, and only temporarily, during handling of the run-duration part of the scene, is it set to `false`—see description of ProcessScene, Section E.3.

The reason for this distinction is that at end of run the user typically wants to display trajectories on a view of the detector, then, at the end of the next event,[8] erase the old and see new trajectories. The visualisation manager messages the scene handler with `ClearTransientStore` just before drawing the trajectories to achieve this.

If the driver supports a graphical database, it is smart to distinguish transient and permanent objects. In this case, every `Add` method of the scene handler must be transient-aware. In some cases, it is enough to open a graphical data base component in `BeginPrimitives`, fill it in `AddPrimitive` and close it appropriately in `EndPrimitives`. In others, initialisation is done in `BeginModeling` and consolidation in `EndModeling`—see `G4OpenGLStoredSceneHandler`. If any `AddSolid` method is implemented, then the graphical data base component should be opened in `PreAddSolid`, protecting against double opening, for example,

```
void G4XXXStoredSceneHandler::BeginPrimitives
(const G4Transform3D& objectTransformation) {
  G4VSceneHandler::BeginPrimitives(objectTransformation);
  // If thread of control has already passed through PreAddSolid,
  // avoid opening a graphical data base component again.
  if (!fProcessingSolid) {
```

If the driver does not have a graphical database or does not distinguish between transient and persistent objects, it must emulate `ClearTransientStore`. Typically, it must erase everything, including the detector, and re-draw the detector and other run-duration objects, ready for the transients to be added. File-writing drivers must rewind the output file. Typically:

```
void G4HepRepFileSceneHandler::ClearTransientStore() {
  G4VSceneHandler::ClearTransientStore();
  // This is typically called after an update and before drawing hits
```

[8] There is an option to accumulate trajectories across events and runs—see commands `/vis/scene/endOfEventAction` and `/vis/scene/endOfRunAction`.

```
  // of the next event.  To simulate the clearing of "transients"
  // (hits, etc.) the detector is redrawn...
  if (fpViewer) {
    fpViewer -> SetView();
    fpViewer -> ClearView();
    fpViewer -> DrawView();
  }
}
```

`ClearView` rewinds (or rewrites) the output file and `DrawView` re-draws the detector, etc. (For smart drivers, `DrawView` is smart enough to know not to redraw the detector, etc., unless the view parameters have changed significantly—see Section E.1.)

Appendix G. Building a drawn scene graph

Some graphics technologies are capable of capitalising on a scene hierarchy. Open Inventor is one such, with its concept of a scene graph. The GEANT4 implementation of the Open Inventor driver makes use of this for the geometry volume hierarchy; not only that, it implements `SoDetectorTreeKit` which makes only the top-most visible; clicking on this volume makes it invisible and reveals its daughters. Thus, the user can reveal successive levels of detail interactively. The `SoDetectorTreeKit`, along with physical properties—colour, transformation, etc.—is designed to added to a `SoSeparator`.

Other examples of drivers that take advantage of the scene hierarchy are the HepRep drivers. HepRep browsers then let the user control visibility, highlighting and other visualisation attributes through control trees that represent this scene hierarchy.

To implement this feature, the developer needs knowledge of the drawn scene graph, i.e. the parent–child relationships of drawn volumes. This is not necessarily the same as the parent–child relationships of the GEANT4 geometry since the user has the option of making volumes invisible (culling). `G4PhysicalVolumeModel` maintains the parent–child path of the actually drawn volumes (see `G4OpenInventorSceneHandler::GeneratePrerequisites`:

```
G4PhysicalVolumeModel* pPVModel =
  dynamic_cast<G4PhysicalVolumeModel*>(fpModel);
if (pPVModel) {
  typedef G4PhysicalVolumeModel::G4PhysicalVolumeNodeID PVNodeID;
  typedef std::vector<PVNodeID> PVPath;
  const PVPath& drawnPVPath = pPVModel->GetDrawnPVPath();
  G4LogicalVolume* pCurrentLV = pPVModel->GetCurrentLV();
```

One way—possibly the simplest—is to rely on the fact that `G4PhysicalVolumeModel` traverses the geometry hierarchy top down, i.e., parent before child. Therefore the developer is assured that a parent will already have been encountered. The `G4LogicalVolume` pointer is enough to identify this parent so Open Inventor keeps a map of `G4LogicalVolume*` and `SoSeparator*`; if the parent is found in the map, the `SoDetectorTreeKit` is added to that `SoSeparator` and the current volume is added to the map:

```
  // Find mother.  ri points to mother, if any...
  PVPath::const_reverse_iterator ri;
  G4LogicalVolume* MotherVolume = 0;
  ri = ++drawnPVPath.rbegin();
  if (ri != drawnPVPath.rend()) {
    // This volume has a mother.
    G4LogicalVolume* MotherVolume =
      ri->GetPhysicalVolume()->GetLogicalVolume();
    ...
    fSeparatorMap[MotherVolume]->addChild(detectorTreeKit);
    fSeparatorMap[pCurrentLV] = fullSeparator;
    ...
```

If there is no mother, the volume is added to the detector root of the scene graph. (Transients—trajectories, etc.—are added to the transient root; it is necessary to keep geometry objects separate from transients so that they can be cleared separately—see Appendix F.)

`G4XXXSGSceneHandler` also implements the above algorithm.

Appendix H. Enhanced trajectory drawing

H.1. Creating a new trajectory model

New trajectory models must inherit from `G4VTrajectoryModel` and implement these pure virtual functions:

```
virtual void Draw(const G4VTrajectory&, G4int i_mode = 0,
                  const G4bool& visible = true) const = 0;
virtual void Print(std::ostream& ostr) const = 0;
```

To use the new model directly in compiled code, simply register it with the `G4VisManager`, e.g.:

```
G4VisManager* visManager = new G4VisExecutive;
visManager->Initialise();
// Create custom model
MyCustomTrajectoryModel* myModel =
           new MyCustomTrajectoryModel("custom");
// Configure it if necessary then register with G4VisManager
visManager->RegisterModel(myModel);
```

H.2. Adding interactive functionality

Additional classes need to be written if the new model is to be created and configured interactively:

- **Messenger classes**
  Messengers to configure the model should inherit from `G4VModelCommand`. The concrete trajectory model type should be used for the template parameter, e.g.:

```
class G4MyCustomModelCommand
: public G4VModelCommand<G4TrajectoryDrawByParticleID> {
...
};
```

  A number of general use templated commands are available in `G4ModelCommandsT.hh`.
- **Factory class**
  A factory class responsible for the model and associated messenger creation must also be written. The factory should inherit from `G4VModelFactory`. The abstract model type should be used for the template parameter, e.g.:

```
class G4TrajectoryDrawByChargeFactory
  : public G4VModelFactory<G4VTrajectoryModel> {
...
};
```

  The model and associated messengers should be constructed in the Create method. Optionally, a context object can also be created, with its own associated messengers. For example:

```
ModelAndMessengers
G4TrajectoryDrawByParticleIDFactory::
    Create(const G4String& placement, const G4String& name)
{
  // Create default context and model
  G4VisTrajContext* context = new G4VisTrajContext("default");
  G4TrajectoryDrawByParticleID* model =
              new G4TrajectoryDrawByParticleID(name, context);
  // Create messengers for default context configuration
  AddContextMsgrs(context, messengers, placement+"/"+name);
  // Create messengers for drawer
  messengers.push_back(new
    G4ModelCmdSetStringColour<G4TrajectoryDrawByParticleID>
                                          (model, placement));
```

```
    messengers.push_back(new
      G4ModelCmdSetDefaultColour<G4TrajectoryDrawByParticleID>
                                          (model, placement));
    messengers.push_back(new
      G4ModelCmdVerbose<G4TrajectoryDrawByParticleID>
                                          (model, placement));
    return ModelAndMessengers(model, messengers);
  }
```

The new factory must then be registered with the visualisation manager. This should be done by overriding the `G4VisManager::RegisterModelFactory` method in a subclass. See, for example, the `G4VisManager` implementation:

```
G4VisExecutive::RegisterModelFactories()
{
   ...
   RegisterModelFactory(new G4TrajectoryDrawByParticleIDFactory());
}
```

Appendix I. Trajectory filtering

I.1. Creating a new trajectory filter model

New trajectory filters must inherit at least from `G4VFilter`. The models supplied with the Geant4 distribution inherit from `G4SmartFilter`, which implements some specialisations on top of `G4VFilter`. The models implement these pure virtual functions:

```
// Evaluate method implemented in subclass
virtual G4bool Evaluate(const T&) = 0;
// Print subclass configuration
virtual void Print(std::ostream& ostr) const = 0;
```

To use the new filter model directly in compiled code, simply register it with the `G4VisManager`, e.g.:

```
  G4VisManager* visManager = new G4VisExecutive;
  visManager->Initialise();
  // Create custom model
  MyCustomTrajectoryFilterModel* myModel =
        new MyCustomTrajectoryFilterModel("custom");
  // Configure it if necessary then register with G4VisManager
  visManager->RegisterModel(myModel);
```

I.2. Adding interactive functionality

Additional classes need to be written if the new model is to be created and configured interactively. The mechanism is exactly the same as that used to create enhanced trajectory drawing models and associated messengers. See the filter factories in `G4TrajectoryFilterFactories` for example implementations.

Appendix J. Creating and using `G4AttDef` and `G4AttValue` objects

`G4AttDef` and `G4AttValue` are C++ implementations of HepRep [4]. They may be used to pass generic information to the visualisation system. Typically they are required in concrete implementations of the methods `GetAttDefs` and `CreateAttValues` in the base classes `G4VTrajectory`, `G4VTrajectoryPoint`, `G4VHit` and `G4VDigi`. Utility classes `G4AttDefStore` and `G4AttCheck` are provided. For an example of their creation, see the concrete class `G4Trajectory`. For an example of their access, checking and use, see `G4VTrajectory::ShowTrajectory`.

The user also may specify `G4AttDefs` and `G4AttValues` in `G4VisAttributes`. The user is responsible for their memory management. `G4AttValues` may be short lived (the `G4VisAttributes` class makes expendable copies when required) but the `G4AttDefs` must be long lived, for example, a data member, or even a static data member, of the user class—see, for example, `ExN04CalorimeterHit`. Alternatively, the user may make use of `G4AttDefStore`, as in `G4Trajectory`.

```
  // Get user G4Atts...
  const std::map<G4String,G4AttDef>* userAttDefs = visAttribs.GetAttDefs();
  if (userAttDefs) {
    const std::vector<G4AttValue>* userAttValues = visAttribs.CreateAttValues();
    ...
    delete userAttValues;  // These must be deleted after use.
  }
  // Get solid's G4Atts created by G4PhysicalVolumeModel...
  G4PhysicalVolumeModel* pPVModel =
    dynamic_cast<G4PhysicalVolumeModel*>(fpModel);
  if (pPVModel) {
    const std::map<G4String,G4AttDef>* solidAttDefs = pPVModel->GetAttDefs();
    if (solidAttDefs) {
      const std::vector<G4AttValue>* solidAttValues =
        pPVModel->CreateCurrentAttValues(
);
      ...
      delete solidAttValues;  // These must be deleted after use.
    }
  }
```

Fig. J.1. How to access G4Atts.

```
void G4VSceneHandler::LoadAtts(const G4Visible& visible, G4AttHolder* holder)
{
  // Load G4Atts from G4VisAttributes, if any...
  const std::map<G4String,G4AttDef>* vaDefs =
    visible.GetVisAttributes()->GetAttDefs();
  if (vaDefs) {
    holder->AddAtts(visible.GetVisAttributes()->CreateAttValues(), vaDefs);
  }

  G4PhysicalVolumeModel* pPVModel =
    dynamic_cast<G4PhysicalVolumeModel*>(fpModel);
  if (pPVModel) {
    // Load G4Atts from G4PhysicalVolumeModel...
    const std::map<G4String,G4AttDef>* defs = pPVModel->GetAttDefs();
    if (defs) {
      holder->AddAtts(pPVModel->CreateCurrentAttValues(), defs);
    }
  }
  ...
```

Fig. J.2. How to load G4Atts into a user object.

G4PhysicalVolumeModel also provides methods GetAttDefs and CreateCurrentAttValues, which apply to the current volume.

If a graphics driver is capable of representing this generic information, it is required to inspect the G4Att pointers of (a) the visualisation attributes, (b) any object that has one of the above abstract interfaces (G4VTrajectory, etc.) and (c) G4PhysicalVolumeModel when relevant. Fig. J.1 shows some example code from a PreAddSolid method.

The class G4AttHolder and utility routine G4VSceneHandler::LoadAtts are provided to facilitate the loading of G4Atts onto an object for future use (picking, etc.). The target object is an object of a class that publicly inherits G4AttHolder—see, e.g., SoG4Polyhedron in the Open Inventor driver. LoadAtts is used as follows:

```
void G4OpenInventorSceneHandler::AddPrimitive(const G4Polyhedron& polyhedron)
{
  SoG4Polyhedron* soPolyhedron = new SoG4Polyhedron(polyhedron);
  LoadAtts(polyhedron, soPolyhedron);
  ...
```

A snippet from G4VSceneHandler::LoadAtts is shown in Fig. J.2. G4AttHolder's destructor ensures proper clean-up of the G4AttValues by deleting them on destruction of the user object.

References

[1] S. Agostinelli, et al., GEANT4, a simulation toolkit, Nuclear Instruments and Methods in Physics Research, NIM A 506 (2003) 250–303. See GEANT4 Web page: http://cern.ch/geant4.
[2] The GEANT4 User Guide for Application Developers, accessible from the GEANT4 web page [1]. Of particular interest and usefulness is Section 7.1, Built-in commands.
[3] J. Perl, Introduction to GEANT4 Visualisation, Stanford Linear Accelerator Center, http://geant4.slac.stanford.edu/Presentations/vis/G4VisIntroduction.pdf.
[4] J. Perl, HepRep: a generic interface definition for HEP event display representables, Web page: http://www.slac.stanford.edu/perl/heprep.
[5] The GEANT4 Software Reference Manual, accessible from the GEANT4 web page [1].
[6] The GEANT4 User Guide for Toolkit Developers, accessible from the GEANT4 web page [1].
[7] HepRApp: HepRep browser Application, http:/www.slac.stanford.edu/perl/HepRApp.
[8] M.C. Coperchio, et al., WIRED4: World-Wide Web Interactive Remote Event Display, Comput. Phys. Comm. 110 (1998) 155. See also: http://wired.freehep.org.
[9] M. Frailis, R. Giannitrapani, The FRED Event Display: an Extensible HepRep Client for GLAST, Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, CA, USA, March 2003. See also http://arxiv.org/abs/cs.gr/0306031.
[10] S. Tanaka, M. Kawaguti, DAWN for GEANT4 Visualisation, in: Proceedings of the CHEP '97 Conference, Berlin (Germany), April 1997.