From the beginning, the Geant4 Visualization System was designed to support several simultaneous graphics systems written to common abstract interfaces. Today, it has matured into a powerful diagnostic and presentational tool. It comes with a library of models that may be added to the current scene and which include the representation of the Geant4 geometry hierarchy, simulated trajectories and user-written hits and digitizations. The workhorse is the OpenGL suite of drivers for X, Xm, Qt, and Win32. There is an Open Inventor driver. Scenes can be exported in special graphics formats for offline viewing in the DAWN, VRML, HepRApp and gMocren browsers. PostScript can be generated through OpenGL, Open Inventor, DAWN and HepRApp. Geant4’s own tracking algorithms are used by the Ray Tracer. Not all drivers support all features but all drivers bring added functionality of some sort. This paper describes the interfaces and details the individual drivers.

Keywords: Geant4; simulation; radiation; modeling; graphics; ray tracing.
1. Introduction

GEANT4 arose out of a desire to take advantage of a new programming paradigm, namely object-oriented programming, to create a toolkit for the simulation of the passage of particles and radiation through matter that could be developed and maintained by physicists, expert in radiation and particle interactions, from around the world. The initiative for a new toolkit came from KEK and CERN in the early 1990s and this led to a research and development project, RD44, funded by CERN from 1994–1998 in the run up to the commissioning of the LHC, in which the paradigm was realized in the C++ programming language. Many other institutes, notably SLAC, also provided support. Many physicists from around the world gave their time and effort as part of their research programmes. GEANT4 was born out of this project in 1999 as an independent collaboration with its own collaboration agreement, providing open code that has found applications in medicine, space and related domains as well as nuclear and particle physics that were its origins. The modular design, which is a natural consequence of object-oriented programming, makes possible the development by many authors in parallel to a well-defined and safe manner, a process that continues to the present day. A more detailed history of GEANT4 can be found in the original general paper. Some more recent developments and applications are described in Ref. 2.

The visualization system was designed to facilitate the development of GEANT4 and its applications. It was written to a basic abstract interface and itself introduced interfaces for multiple drivers. This was first described in Ref. 3 and implementation details are given in the Toolkit Developers Guide. The current paper brings this up to date and describes progress since then.

A good practical introduction to the GEANT4 Visualization System is given in Ref. 5.

2. Overview of the GEANT4 Visualization System

Every application that needs to use the GEANT4 Visualization System must instantiate a G4VisManager and register visualization drivers as desired from those available on the computing platform. (This is done automatically by building the application through the GEANT4 build system and G4VisExecutive as described below.) The application communicates with the visualization manager through basic abstract interfaces, also described below.

The GEANT4 Visualization Manager has the ability to build “scenes” (G4Scene objects) with any number of geometrical (detector) components, axes, annotations and (at the end of event or run) the potential to draw particle trajectories, hits (representations of effects, for example, energy deposit, of particles) and digitizations (ultimate signals from sensitive components—“digits” for short). It defines, through the G4SceneHandler interface, “scene handlers” that translate the scene into messages for a particular graphics system and, through the G4Viewer interface, “viewers” that render to the final device (screen, file or terminal). Any scene can
be associated with any scene handler and viewed with any viewer. A scene handler-viewer(s) combination is referred to as a driver. One may instantiate any number of drivers of any type, each with any number of viewers, and switch between them.

It is a natural consequence of object-oriented design that any driver that conforms to these interfaces can be accessed by the visualization manager and we take advantage of this to write multiple drivers with different characteristics and qualities—for example, OpenGL and Open Inventor for fast drawing (with varying degrees of interactivity according to the computing platform), DAWN for high quality PostScript output, HepRepFile for scene browsing, VRML for virtual reality, RayTracer for photorealistic images, gMocren for medical images and ASCITree for a text dump.

The Geant4 Visualization Manager keeps a list of scenes, scene handlers and viewers. There is always a current viewer serviced by its scene handler with its scene. Drawing requests made by the application or reissued by the visualization manager are always made to the current viewer. If a scene is changed, or on user request, all views of that scene are rebuilt.

The Geant4 Visualization System is implemented as a “plug-in”. It may use a part of the tool kit itself or by the Geant4 user application or by the toolkit itself through the basic abstract interfaces in a protected way, as described below.

The Geant4 Visualization System is currently being used across a wide range of Geant4 applications in high energy physics, nuclear physics, space and medicine.

3. The Basic Abstract Interfaces

Following our object-oriented design, any Geant4 Visualization System Visualization Manager must implement the G4VVisManager interface. This is the working interface for all drawing messages. As one can see from Fig. 1, it is generously endowed with possibilities.

The Geant4 visualization manager interface is available to all Geant4 code whether in a Geant4 user application or in the toolkit itself. However, the coder must check that a concrete implementation actually exists and avoid using the interface if not; for example:

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

This allows one to build a Geant4 application without a concrete implementation, for example for batch production, even if such code remains in the application or the toolkit (which it certainly does).

A second interface—G4VGraphicsScene (see Fig. 2)—is private to the toolkit. It is not available for application developers. It is used by the toolkit—geometry and modeling in particular. The visualization manager is expected to initiate its use by providing a reference to a concrete implementation.
In summary, a visualization system is required to implement two interfaces—
G4VisManager (see Fig. 1) and G4VisualScene (see Fig. 2). The latter is, in
fact, intended to represent the “scene handler”. Objects that are passed to it (G4Box
objects, etc.) are required to be turned into visible renderings. Quite how this is
done is up to the visualization system.

4. The Geant4 Visualization System

So now we are in a position to describe the Geant4 Visualization System, which
is the implementation of the above interfaces that is distributed with the Geant4
toolkit.

Geant4 defers to the application developer the decision of which external
graphics packages should be required. Accordingly, it is the application developer’s
responsibility to declare which graphics drivers should be made available to the end
user. He or she makes these declarations by instantiating a `G4VisManager` (written to the `G4VisManager` interface) and registering appropriate graphics drivers. The Geant4 distribution provides a general purpose sub-class called `G4VisExecutive`, which collaborates with the Geant4 build system to offer all available drivers and which is used in all Geant4 examples. This is shown in Fig. 3.

`G4SceneHandler` is written to the `G4GraphicsSystem` interface and itself is the base class for graphics-library-dependent concrete scene handlers, for example,
for OpenGL. The system is capable of handling multiple drivers, i.e., multiple scene handlers and viewers.

4.1. Visualization of touchables

GEANT4 has a layered geometry structure, in which solids define shapes, logical volumes add material information to solids and physical volumes place a given logical volume in space. To make efficient use of memory, GEANT4 provides mechanisms whereby a single physical volume may have more than one placement, using mechanisms called Replica placement or Parameterized placement. Whether a physical volume has only one placement, or many placements, each placement is called a touchable.

The visualization system must deal in touchables, and it is the job of G4PhysicalVolumeModel to roll out the GEANT4 geometry into touchables. To avoid being overwhelmed, it is possible to specify starting points in the geometry hierarchy, such as specific sub-detectors, and limit the depth of descent so as to avoid too much detail. One can also control, for example, the visibility with G4VisAttributes (see Sec. 4.5); invisible touchables may then be suppressed (culled).

It is possible to edit the G4VisAttributes of logical volumes with /vis/geometry/ commands and to modify the G4VisAttributes of individual touchables with /vis/touchable/ commands. The latter can also be done interactively in the OpenGL Qt viewer.

4.2. Visualization of transients

"Transients" is a term for entities that appear for a limited time on a “permanent” background, for example, particle trajectories in a detector. The GEANT4 Visualization System distinguishes these; smart drivers may take advantage. For example the OpenGL “stored” system (see Sec. 6.1.1) keeps separate lists in its graphical database for transient and permanent objects, and the GEANT4 Visualization Manager clears and rebuilds the former without having to rebuild the latter. This makes it very efficient for displaying event after event on a fixed detector image.

The GEANT4 Visualization System allows one to display particle trajectories, hits and digits event by event or to accumulate them, one on the other, until the end of run.

This is fine for drivers with their own graphical database, but in such circumstances, not-so-smart drivers have to redraw both permanent and transient objects, with a noticeable performance degradation. Moreover, they may not even have the ability to remember permanent objects, let alone transients. For these drivers, the GEANT4 Visualization System keeps a memory of permanent objects (for example, it may always revisit the geometry hierarchy to rebuild a detector image) and also keeps a memory, to a limited but considerable extent, of transient objects such as particle trajectories, hits and digits that are stored in simulated events. With this
The GEANT4 Visualization System

feature, described in Sec. 4.3, all drivers, including the not-so-smart, can recover the transients and emulate the superposition of transients on permanents, a switch of viewpoint or even a switch of drivers.

Transients that are generated from user code will not be recoverable in this way, unless the user encloses the code in a User Vis Action—see Sec. 4.6.

4.3. **Event storing**

The GEANT4 Visualization Manager asks the Run Manager to keep events so that they may be accessed to redraw, say, the particle trajectories, hits and digits in a view. This is particularly useful for drivers that do not have their own graphical database or when switching from one driver to another. The default behavior is to keep the last 100 events, but the number may be adjusted to take into account available memory; also user code can provide algorithms to specify which events should be kept.

4.4. **Trajectory modeling and filtering**

An extensive set of models and filters for trajectories drawing is available. They control the color and visibility by particle type and the visibility by momentum or detector volume, for example. Some commands are part of the start-up script shown in Fig. 8. Models may also select whether the trajectory is drawn as a line, as step points, or as both.

A similar but less extensive set is available also for user-defined hits and digits. A user may add attributes to trajectories, hits and digits and filter them at drawing time with `/vis/filtering/` commands.

4.5. **Attributes**

Every drawable entity is assigned visualization attributes `G4VisAttributes` either by the user or from a modifiable default set. They include visibility (i.e., drawn or not if culling is active), color, line width, line style and the ability to “force” some modes of drawing, such as wireframe or with surfaces, regardless of the general user request.

Some entities have additional attributes in the form of `G4AttDef` and `G4AttValue` pairs, which are strings to be interpreted as text, integers, doubles or three-vectors, with or without dimension (units), following the HepRep attribute design. For example, trajectories acquire particle type, momentum, volume name and process type. A full list for `G4RichTrajectory` objects is shown in Fig. 4. (Note that this is memory-consuming; by default trajectories are stored as `G4Trajectory` objects with fewer attributes.)

Touchables also get attributes—see Fig. 5.
It is these attributes that are selectable with /vis/filtering/ commands, as mentioned above, and that may be dumped by picking with pick-sensitive drivers—OpenGL, OpenInventor and HepRApp (the browser for HepRepFile)—see below.
4.6. **User vis actions**

As we have said, a user may draw to the current viewer at any time with C++ code written to the basic abstract interface (see Sec. 3). However, the drawing will belong to the set of unrecoverable “transients” (see Sec. 4.2) and will not be preserved on change of view or change of driver. A better strategy is to write a User Vis Action, a class that inherits `G4UserVisAction`, instantiate it and register it with the Geant4 Visualization Manager. This way the visualization manager can invoke the code repeatedly as required to give the drawn objects as sort of permanence.

```cpp
class StandaloneVisAction: public G4UserVisAction {
  virtual void Draw() {
  
  
  }
};

void StandaloneVisAction::Draw() {
  G4VisManager* pVisManager = G4VisManager::GetConcreteInstance();
  if (pVisManager) {
  // Simple box...
  pVisManager->Draw(GBox('box',2*m,2*m,2*m),
    G4VisAttributes(G4Colour(1,1,0)));
  
  // Boolean solid...
  G4Box boxA('boxA',3*m,3*m,3*m);
  G4Box boxB('boxB',3*m,3*m,3*m);
  G4SubtractionSolid subtracted('subtracted_boxes',boxA,boxB,
    G4Translate3D(3*m,3*m,3*m));
  pVisManager->Draw(subtracted,
    G4VisAttributes(G4Colour(0,1,1)),
    G4Translate3D(-6*m,-6*m,-6*m));
  }
}
```

Fig. 6. A User Vis Action that gives drawn objects a permanence.

![User Vis Action Example](image)

Fig. 7. A result of the code shown in Fig. 6. (This figure was produced with `/vis/ogl/printEPS` from an OpenGL stored Xm window).
J. Allison et al.

A few examples are included in the GEANT4 distribution. examples/extended/visualization/userVisAction shows how to implement a user-defined logo. examples/extended/visualization/standalone shows, as the name implies, how the GEANT4 libraries can be used as a “standalone” multi-driver graphics library. Figure 6 shows the code that results in Fig. 7. The reader will notice that one may use the GEANT4 geometry, including, for example, the Boolean solids (G4SubtractionSolid, etc.).

5. The User Interface

A user may code to the basic abstract interface, G4VVisManager, or, more usually, use an extensive set of visualization commands (see Sec. 5.1) through the GEANT4 User Interface. GEANT4 offers several user interfaces, ranging from dumb terminal to graphical. In general, any user interface will work with any visualization driver, except in the case of Qt7 (which offers the most advanced interactivity, and for which the graphics window is closely coupled to the user interface).

A user may also read commands from a file by typing “/control/execute <filename>” or by programming the reading of a file using G4UImanager::ApplyCommand(). This is typically to be found in the examples distributed with the GEANT4 toolkit.

5.1. Visualization commands

The visualization commands are too numerous to detail here. A full description of all commands is available in the Application Developers Guide.

For example, a user would instantiate a user interface and then issue commands, such as “/vis/open OGL” and “/vis/drawVolume” to get an image of the detector. Figure 8 shows a typical start-up script. After reading this file, “/run/beamOn 10”, for example, would run the simulation and draw particle trajectories from 10 events.

6. The Drivers

Over the years we have developed, at the latest count, 14 drivers of various sorts (or upward of 20 if one counts all the OpenGL variants and DAWN and VRML variants). A user may draw to the basic abstract interfaces, either in C++ code or, more usually, via visualization commands (see Sec. 5.1) through a user interface, and expect it to be rendered in one of a number of different ways: to a computer screen (graphics drivers, Sec. 6.1); or to a file for subsequent browsing (file-writing drivers, Sec. 6.2). There is also a category of pseudo-drivers (Sec. 6.4).

Some drivers support picking, i.e., clicking on an item in the graphics window pops up a window of information or dumps a print-out of attributes to standard output. In this category are: HepRepFile; OpenGL X11; OpenGL Qt; Open Inventor.
The Geant4 Visualization System

Fig. 8. A typical start-up script.
6.1. Graphics drivers

The workhorse of the Geant4 Visualization System is the set of OpenGL drivers. We also have a graphics driver for Open Inventor and a driver, RayTracer, that uses Geant4’s own tracking algorithms to produce a ray-traced image.

6.1.1. OpenGL

A particular feature of OpenGL is that one may store GL commands in a “display list” that may be efficiently rendered by a graphics processing unit. We recommend this “stored mode” as the default option to get good performance for rotating, zooming, etc., without having to revisit the Geant4 kernel. However, a complex detector or complex event structure can overwhelm a computer system, so we offer an “immediate mode” whereby objects are rendered directly to the screen, but to change viewpoint, for example, the graphics system revisits the Geant4 kernel for information about the scene and makes all the coordinate calculations afresh, which is much slower.

The OpenGL drivers have various degrees of interactivity depending on the availability of advanced graphics libraries. The simplest—OpenGL X11 (Unix) and
Win32 (Microsoft Windows)— are passive. For example, to change the viewpoint one must issue a command, e.g., "/vis/viewer/set/viewpointThetaPhi 30 30 deg". With Motif libraries one can build OpenGL Xm; the viewer provides some interactivity via pull-down menus, including rotation and zoom. The most sophisticated one is the OpenGL Qt driver, which offers a huge amount of interactivity, including rotation, pan and zoom, picking, drawing style, projection style, etc. The Qt user interface, G4UIQt, which must be used with it, includes an interactive help system and an interactive portrayal of the scene, including the geometry hierarchy, through which one can change the color and visibility of individual screen objects. Figure 9 shows a typical screen shot.
The OpenGL driver also implements transparency and cutaways. A user may save the view to file in EPS format with /vis/ogl/printEPS.

6.1.2. Open Inventor

The Open Inventor drivers for Xt (Unix) and Win32 (Microsoft Windows) also provides good interactivity. Figure 10 shows a typical screen shot. At present, this driver cannot render “2D” objects such as nonmoving descriptive text. It may be worth mentioning that as in OpenGL the view may be saved in EPS format via a menu button on the viewer.

6.1.3. Ray Tracer for X

The Ray Tracer can only render geometry. Figure 11 shows a typical screen shot. The Ray Tracer can also produce JPEG files directly—see Sec. 6.2.5.

Fig. 11. A screen shot of a Ray Tracer viewer for X11.
6.2. File-writing drivers

6.2.1. HepRepFile

HepRepFile allows the current scene (geometry, event or collections of events) to be rendered to an XML file in the HepRep format. The file can then be read into a HepRep browser such as HepRApp. Because the HepRep file contains a full 3D description of the scene, augmented with full physics attribute data (attributes of the touchables, trajectories and hits), the HepRApp user can then rotate, pan, zoom, change projection styles, pick to view attributes and filter graphics based on attributes. Resulting images can export a wide variety of graphics formats (gif, pdf, etc.).

Figure 12 shows a screen shot of HepRApp browsing a file produced by HepRepFile.

6.2.2. DAWNFILE

Similarly, DAWNFILE produces a file suitable for browsing with DAWN. DAWN is capable of sophisticated hidden line and hidden surface removal and produces very high quality images in encapsulated PostScript format–see Fig. 13.

One can generate cutaways with DAWNCUT. A useful spin-off is DAVID, which is a volume overlap detection application.

![Fig. 12. A screen shot of HepRApp browsing a file produced by HepRepFile.](image-url)
6.2.3. **VRMLFILE**

Similarly, VRML2FILE produces a file suitable for browsing with a VRML browser. Note that VRML1 is still available.

6.2.4. **gMocrenFile**

gMocren\(^4\) is used typically to visualize radiation therapy dose data. Figure 14 shows a rendering with a gMocren viewer of a file that has been created with gMocrenFile after a simulation run.

6.2.5. **Ray Tracer jpeg-writer**

The Ray Tracer can produce a JPEG file, either from its X11 version (described above, Sec. 6.1.3, and instantiated with “/vis/open RayTracer\(X\)” or from this version that needs no graphics library, instatiated within “/vis/open RayTracer”.

6.2.6. **Encapsulated PostScript**

For completeness we reiterate the possibility, mentioned above, of saving a view with any variant of OpenGL or Open Inventor driver to file in EPS format with the visualization command /vis/ogl/printEPS. This can be done even without a user interface (see Sec. 5) by using the **ApplyCommand()** method of the User Interface...
Manager. The DAWNFILE and HepRepFile drivers can also be used in this way to produce files that can be browsed to produce an EPS file.

6.3. **BSD socket drivers**

There are versions of the DAWN and VRML drivers that can communicate with their respective browsers though a socket mechanism. The browsers have to be launched first and suitable socket numbers chosen.

6.4. **Pseudo-drivers**

We have taken advantage of the way the visualization system works to write a useful pseudo-driver. The visualization system renders the scene to the “Tree” driver and dumps useful information.

6.4.1. **ASCII Tree**

At the moment, this is the only tree driver. It dumps the geometry tree to standard output and lists the physical volume name and optionally also the logical volume
name, the solid name, its volume, its density, its mass excluding daughters and the total mass of the top physical volume. Figure 15 shows typical output.

7. Summary
We have developed a visualization system that is versatile, powerful and extensible. It was designed to meet the needs of Geant4 users. It supports several drivers over various graphics libraries, including OpenGL, Open Inventor and Geant4’s own tracking library (Raytracer) and can write files for various browsers: DAWN, VRML and HepRAp. It handles the Geant4 geometry hierarchy through a modeling library and can draw particle trajectories from stored events in a variety of ways.

Acknowledgments
We wish to acknowledge the team of Geant4 collaborators who have made this work possible. In particular, we wish to acknowledge the contribution of Satoshi Tanaka to the DAWN, VRML and gMocren drivers and to the DAWN and gMocren browsers. J. Allison wishes to acknowledge the support of The University of Manchester and Geant4 Associates International Ltd. L. Garnier wishes to acknowledge the support of Laboratoire de l’Accélérateur Linéaire, Orsay. This work is supported in part by the U.S. Department of Energy under contract number DE-AC02-76SF00515.
The Geant4 Visualization System

References


7. The Qt Project, an open source cross-platform application and UI framework, Available at http://qt-project.org/.

8. The Geant4 User Guide for Application Developers, accessible from the Geant4 web page. Of particular interest and usefulness is Section 7.1, Built-in commands.

9. For example, one may obtain Motif libraries from the Open Group, Available at http://www.opengroup.org/openmotif/.


