Visualization in GEANT4 — An Overview and User Guide Version 0-5

John Allison and Andrew Walkden
University of Manchester

Satoshi Tanaka
University of Fukui

14th July 1998

CVS $Id: VisInG4.tex,v 2.1 1998/07/04 16:12:49 allison Exp $
CVS tag: visualization-0-5

Abstract

The Visualization System of GEANT4 consists of a Visualization Manager, which processes user commands, and a Visualization Interface, a "thin" interface which defines the functionality required of the graphics systems by the Visualization Manager and the GEANT4 kernel. An installer may write drivers for his or her own graphics system or use drivers which come ready-to-use with the GEANT4 toolkit; drivers are instances of C++ classes derived from abstract classes in the Visualization Interface. There exist facilities for debugging, development, communication and publicity which will evolve as effort accumulates and graphics systems become more powerful. This document describes those facilities and is intended to act as a GEANT4 Visualization User Guide.

1 Introduction

Visualization is an essential part of a detector simulation program. It is used to check the geometrical model of the detector and to gain confidence in the physics processes that are being simulated. The Visualization System also helps to evolve detector design. And, not least, visualization is a communication and publicity tool.

Past versions of GEANT, notably GEANT3[1], offered facilities for the user to check his or her work by drawing individual detector components, sub-detector assemblies and whole detectors. It was possible to eliminate hidden lines and shade surfaces, but the predominant style was wireframe and sections (cuts). For example, it was common to superimpose particle steps and detector hits on a section of the detector as a way of confirming the correct behaviour of the program and of appreciating the tracking and interaction processes which were being simulated.

Similar visualization facilities are/will be provided in GEANT4. However, GEANT4 has a look and feel which is considerably different to GEANT3's. It exploits advances in computer equipment and graphics software, such as X Windows and OpenGL. There are/will be new and more sophisticated facilities such as smoothly changing viewpoints, interactive cutaways and other interactive possibilities which rely on the greater computer system performance available today. We hope the Visualization Interface will prove to be flexible enough to accomodate new facilities and new graphics systems as they become available.

In fact, the Visualization Interface is an interface which, by design, is "thin", that is, minimal and abstract. Apart from memorising the scene definition ("Scene Data") for each scene and "View Parameters" for each view, and containing code which "knows" how to interpret GEANT4 geometry objects, it consists solely of "virtual methods" which have to be implemented by the graphics system installer when writing the drivers for the graphics system. This is an example of GEANT4's "toolkit" approach, and it relies heavily on object oriented technology.

In designing the Visualization System, our guide has been the original User Requirements Document; the next section describes how well we have satisfied, or surpassed, them. Subsequent sections outline the design of the Visualization System and the way of communicating with it to obtain drawings on the computer screen or on hard copy. Details are left to the Appendices.

2 User Requirements

The design of the Visualization System has been driven by a set of user requirements which have been extensively propagated and discussed. As far as the Visualization System is concerned, they are summarised in Appendix A.

Note at 14th July 1998:

For the alpha release, not all the requirements are implemented. There is no "picking" or, indeed, anything which requires feedback from the graphics system to the GEANT4

[1]GEANT3 currently exists in version 3.21, which will probably be the last.

Visualization Manager or kernel. On the other hand, the alpha release offers multiple simultaneous graphics systems and multiple view manipulation.

Other requirements which are yet to be implemented are: sections, cutaways, track hierarchy display and geometry hierarchy (DTREE) display.

The graphics system drivers that are currently distributed with GEANT4 are:

- The Fukui Renderer (DAWN) [1], for high quality graphics and hardcopy (PostScript).
- OPACS [2], an X-Windows-based system in the public domain.
- OpenGL.
- Open Inventor.

The DAWN and OPACS packages are available with the GEANT4 distribution in the form of source code and make files. the OpenGL and Open Inventor libraries are available commercially, although a free OpenGL, known as Mesa [3], works well. CERN has purchased licences, but outside institutes have to provide their own [4].

Note at 14th July 1998:

Ray tracing, using GEANT4's own tracking algorithms, will be added later.

3 Installation

GEANT4 is a "toolkit". The installer at each site decides which features are required, and compiles and builds appropriate libraries for the user. In particular, he (or she) decides which graphics systems to use. He chooses from the graphics system drivers supplied with GEANT4 depending on which libraries and packages he has available. Also, if he so wishes, he can write drivers for a graphics system, by meeting the specification defined by the Visualization Interface. This is described in detail in The GEANT4 Visualization Manager and Graphics System Interface Specification and Reference [5], (hereafter known as The Reference). It involves writing C++ classes publically derived from the Visualization Interface abstract base classes and implementing the virtual functions therein specified.

4 Detector Simulation

The user then compiles code which describes his geometry and writes and compiles user action functions to accumulate hits and digitisations (simulated signals) and, if he wishes, to draw them, as described below (Appendix D). He then links everything together to make a GEANT4 executable.

5 Running GEANT4

For a particular GEANT4 executable, through the GEANT4 User Interface, which may be command-line-based or graphical, the user issues commands (those relevant to visualization are described in detail in Appendix B). For example, he may draw a detector component. Typically, he then initiates the simulation of an event which, if the appropriate user action routines have been written, will also draw the steps, hits or digitisations. The next section describes features of the Visualization Interface which the user needs to understand in order to make good use of GEANT4 graphics.

6 Features and Concepts of GEANT4 Visualization

6.1 The Current View, Scene and Graphics System

The user may open several views of several scenes with more than one graphics systems at the same time. Thus there might be more than one windows open. So the Visualization Manager needs to have the concept of the "Current View", that is, the one that the drawing commands will affect. The "Current Scene" is the scene corresponding to the Current View. And, of course, the "Current Graphics System" is the corresponding graphics system. The Visualization Manager maintains pointers to the Current View, Scene and Graphics System.

The user may create views and switch between them. He can also querie the view and list available views and graphics systems.

6.2 Scene and Scene Data

A "scene" is defined by a set of "stored" GEANT4 objects which the user wishes to view. Together with their associated characteristics, they comprise the "Scene Data". The GEANT4 objects which qualify are: physical volumes, trajectories (if they are stored), hits and digitisations, that is, anything that is guaranteed to be available at the end of simulating an event (that is what is meant by "stored").

In addition, objects can drawn in so-called "transient" mode; such objects are not part of the scene, but are superimposed on individual views of it — see Section 6.6. In particular, the transient mode is the only way to draw non-stored objects such as steps and non-stored trajectories.

Each scene is represented by a C++ object which is an instance of a graphics-system-dependent class derived from `G4VScene`. Each such C++ object keeps its own Scene Data encapsulated in a `G4SceneData` object. Details are given in The Reference [5] but the User Interface Commands to manipulate the scene are described in Section 8 and Appendix B.

6.3 View and View Parameters

A "view" is a particular graphical rendering of a scene. Normally this is a window on the computer screen, but it may be a piece of paper (hardcopy). The view is defined by the scene and additional "View Parameters", such as viewpoint direction and lighting. The View Parameters also include drawing style, for example, wireframe or solid, but this may be overriden for individual GEANT4 scene objects if the user "forces" it by specifying the appropriate Visualization Attribute when he writes the geometry definition code (see Section C.2), or, perhaps, by a user command.

Each view is represented by a C++ object which is an instance of a graphics-system-dependent class derived from `G4VView`. Each such C++ object keeps its own View Parameters encapsulated in a `G4ViewParameters` object. Details are given in The Reference [5] but the User Interface Commands to manipulate the view are described in Section 8 and Appendix B.

6.4 The Standard View and All That

In GEANT4 visualization, we have the concept of a "Standard View". This is the view when the complete set of objects being viewed is comfortably in view from any viewpoint. It is defined by the "Bounding Sphere" of "visible" objects when initially registered in the scene, and the View Parameters.

There is also the "Standard Target Point", which is the centre of the Bounding Sphere (note that these belong to the scene, and are stored in the Scene Data). The "Current Target Point", initially set to the Standard Target Point, is incremented by the "dolly" and "zoom" commands, and can be reset to the Standard Target Point with the `/vis~/camera/reset` command.

Also, the "Standard Camera Position" is the "Standard Camera Distance" along the Viewpoint Direction vector from the Standard Target Point. The Standard Camera Distance is the radius of the Bounding Sphere divided by `fFieldHalfAngle`. It is not stored explicitly because of the singularity at `fFieldHalfAngle` = 0, which implies parallel projection.

Similarly, the "Current Camera Position" is the "Current Camera Distance" along the Viewpoint Direction vector from the Current Target Point. The Current Camera Distance is given by the formulae below, but note that it can be negative, meaning that the camera has moved *beyond* the Current Target Point, which is conceptually possible, but which might give some problems when setting up the view matrix — see, for example, `G4OpenGLView::SetView ()`.

All views are expected to keep the "Up Vector" vertical.

Finally, the view is magnified by the "Zoom Factor" which is reset to 1 by the `/vis~/camera/reset` command.

Note: the camera movements pan, dolly and zoom, are currently coded in `G4VisManMessenger.cc`.

The algorithms for calculating various useful quantities from the View Parameters are encapsulated in the functions described in Appendix E.

6.5 Current Scene Data and View Parameters

The Visualization Manager keeps a memory of the Scene Data and View Parameters — the so-called "Current Scene Data" and "Current View Parameters" — which usually, but not always, correspond to those of the Current View.

However, the user can create a new view, or select a different existing view. Thus the Current View will change, but not the Current Scene Data and View Parameters. The memory in the Visualization Manager means that if he now issues a draw command, the picture in the new window will be a view of the previous scene with the previous View Parameters, rendered by the same or different graphics system. (Transient objects, such as steps, are not memorised in the Scene Data, so are excluded from this procedure.)

Thus a draw command forces the Current View to have the same View Parameters as those stored in the Visualization Manager. The user may also do the opposite, that is, select a view and request that the Current Scene Data and View Parameters be copied, that is, made to correspond to those of the Current View. A subsequent change of view, followed by a draw, will therefore copy the view.

In these ways, a user can have several views of the same or different scenes and copy them (except transient objects) from view to view, and graphics system to graphics system. In particular, he may experiment to find a suitable view using wireframe for speed, then copy, change drawing style to solid, to get a different view, or copy to a high quality renderer.

6.6 The Transient Drawing Mode

Apart from objects which are specified in the scene, which, as described above, are "remembered" by the scene and can therefore be re-drawn or copied and drawn at any time, the user may draw objects in "transient mode". These may include non-stored objects such as steps and non-stored trajectories, or anything a user desires. Of course, they must be converted to something the Visualization Manager can understand — see Appendix D.

The user superimposes such objects on the Current View view by writing C++ code in the appropriate user action routine. An example is given in Appendix D. He cannot copy them from view to view. They might or might not disappear if he attempts, for example, to change the viewpoint direction, depending on whether the graphics system keeps its own memory of drawn objects, for example in a display list; that is, the behaviour is graphics-system dependent.

Although hits and digitisations, etc., are stored, and thus are eligible for inclusion in the scene, the user may choose to draw them transiently by writing code in the appropriate use action routine. In that case, of course, they cannot be copied from view to view, etc.; in other words, they will then behave like transients.

7 Visualization Attributes

Most drawable objects can have Visualization Attributes — visibility, colour, etc. What these are and how to set them is described in Appendix C.

8 User Interface

Like the graphics systems, the User Interface can be chosen by the installer. It can be graphical or command-line oriented. The user manipulates the scenes and views through the User Interface; the currently available visualization commands are listed in Appendix B.

9 Conclusion

Some basic visualization facilities are available in the alpha release of GEANT4. Details are given in the Appendices. Try them, enjoy them or be frustrated by them — either way, let us know what you think.

A User Requirements

The specific user requirements for visualization as set out in the requirements document are (to quote):

UR 6-1. The detector setup can be visualized in its entirety or in subsections or by individual component.

UR 6-2. The individual geometrical entities representing a detector can be visualized and their parameters displayed.

UR 6-3. Particle trajectories can be visualized and each step of the particle tracking can be visualized.

UR 6-4. The detector response in sensitive elements of the detector can be visualized.

UR 6-5. The visualization system is capable of navigating the genealogy relationships of information in an event, for example, highlighting all hits from a given track, all tracks from a given parent track, etc.

UR 6-6. The visualization is capable of showing the geometry structure, that is, the genealogy relationships of the geometry.

UR 6-7. There is a built-in visualizer of reasonable sophistication. However, the toolkit can provide a straightforward way of connecting to other visualization systems so that the provider can build a visualizer of his/her choice into the framework. Some specific interfaces are provided.

To these, the visualization working group have added:

1. Unless the user decides otherwise (for example, for performance reasons) the drawings are faithful graphics representation, that is, there are no unnecessary approximations (for example, polygons for circles) worse than the device resolution.
2. The user can view several different scenes simultaneously and a given scene from several viewpoints.
3. The Visualization Interface can support more than one graphics system concurrently.
4. "Transient" objects such as steps and tracks can be drawn on an existing view of "permanent" objects (such as detectors) in a transient way (drawn and erased) or accumulated.
5. Remote graphics can be supported, that is, a graphics renderer will be made available that can be spawned and run on a computer different to the one running the GEANT4 kernel.
6. Graphics actions (for example, mouse movement to produce camera movements) should be possible in parallel with other GEANT4 actions or commands.
7. The user can draw sections or cutaways of the detector.

B Visualization Commands Available through the GEANT4 User Interface

GEANT4 commands are structured hierarchically in command directories and subdirectories. At the top or root level, there exists, amongst others, the `/vis~/` command directory; all Visualization Commands are in that command directory or its subdirectories.

Under `/vis~/` there are the following command subdirectories:

Command	Arguments	Guidance
`/vis~/camera/`		...menu of camera commands.
`/vis~/clear/`		...menu of clear commands.
`/vis~/copy/`		...menu of copy commands.
`/vis~/create_scene/`		...menu of scene creation commands.
`/vis~/create_view/`		...menu of view creation commands.
`/vis~/delete/`		...menu of deleting possibilities.
`/vis~/draw/`		...menu of drawing possibilities.
`/vis~/lights/`		...menu of lights commands.
`/vis~/print/`		...menu of printing possibilities, also controlled by "verbose" parameter.
`/vis~/refresh/`		...menu of refresh commands.
`/vis~/set/`		...menu of set commands.
`/vis~/show/`		...menu of show commands.
`/vis~/expert/`		...menu of expert commands.

The commands available under the `/vis~/camera/` command subdirectory are:

Command	Arguments	Guidance
`/vis~/camera/reset`		Resets dolly, pan and zoom. Regains "Standard View".
`/vis~/camera/dolly`	distance	Moves the camera in by this distance. Reset with `/vis~/camera/reset`.
`/vis~/camera/orbit`	N_{frames}, $\Delta\beta$	Orbits the scene about the up-vector, lights fixed to the scene. Draws N_{frames} frames, the camera rotated $\Delta\beta$ about the up-vector each frame.
`/vis~/camera/pan`	right, up	Moves the camera right and up by these amounts. Reset with `/vis~/camera/reset`.
`/vis~/camera/projection_style`	choice	Projection style (orthogonal, perspective).
`/vis~/camera/spin`	N_{frames}, $\Delta\beta$	Spins the scene about the up-vector, lights fixed to the camera. Draws N_{frames} frames, the scene rotated $\Delta\beta$ about the up-vector each frame.
`/vis~/camera/viewpoint`	θ ϕ	Set direction from target to camera as θ, ϕ (in degrees). Also changes lightpoint direction if lights are set to move with camera.
`/vis~/camera/window_size_hint`	size	Size in pixels (square window 'size $\times$ size'). Affects future create_view commands (hint only).
`/vis~/camera/zoom`	factor	Magnifies by this factor. Reset with `/vis~/camera/reset`.

The commands available under the `/vis~/clear/` command subdirectory are:

Command	Arguments	Guidance
`/vis~/clear/scene`		Clears current scene and scene data and marks all its views as needing refreshing (but does not clear them).
`/vis~/clear/view`		Clears visible window of current view.
`/vis~/clear/view_and_scene`		Combination command.

The user can also copy data and parameters pertaining to the current view into the Visualization Manager's memory, and then transmit them to other views, even views belonging to other graphics systems, by subsequently selecting the other view and drawing. The net effect is to copy the data from the source view to the target view. So the order of commands is:

```
/vis~/set/view <source>
/vis~/copy/all
/vis~/set/view <target>
/vis~/clear/view
/vis~/draw/current
```

The following commands enable this, but *be warned — these commands overwrite the current scene data and/or the current view parameters.*

Command	Arguments	Guidance
`/vis~/copy/all`		Copy all data of current scene and view into current memory. WARNING: this overwrites the current scene data and current view parameters.
`/vis~/copy/scene`		Copy scene data of current scene into current scene data. WARNING: this overwrites the current scene data.
`/vis~/copy/view`		Copy view parameters of current view into current view parameters. WARNING: this overwrites the current view parameters.

The commands available under the `/vis~/create_scene/` command subdirectory are:

Command	Arguments	Guidance
`/vis~/create_scene/clear`		Same as `/vis~/clear/scene`.

The commands available under the `/vis~/create_view/` command subdirectory are:

Command	Arguments	Guidance
`/vis~/create_view/new_scene`		Creates a new scene and a new view; both become current.
`/vis~/create_view/new_view`		Creates a new view of current scene; new view becomes current view.
`/vis~/create_view/new_graphics_system`	choice	Creates a new scene and a new view of a new graphics system; all become current.

The commands available under the `/vis~/delete/` command subdirectory are:

Commands	Arguments	Guidance
`/vis~/delete/scene`		Deletes the current scene and its views.
`/vis~/delete/view`		Deletes the current view.

The commands available under the `/vis~/draw/` command subdirectory are:

Commands	Arguments	Guidance
`/vis~/draw/current`		Draws the current scene in the current view using the current view parameters.
`/vis~/draw/axes`	x_0, y_0, z_0, length, unit	Draws axes at (x_0, y_0, z_0) of given length.
`/vis~/draw/text`	x, y, z, unit, font_size, x_offset, y_offset, text	Draws text at (x, y, z) with given parameters. Font size and offsets in pixels.

The commands available under the `/vis~/lights/` command subdirectory are:

Command	Arguments	Guidance
`/vis~/lights/move_with_camera`	true/false	Lights move with change of viewpoint.
`/vis~/lights/direction`	θ ϕ	Set direction from target to light as θ, ϕ (in degrees).

The commands available under the `/vis~/print/` command subdirectory are:

Commands	Arguments	Guidance

The commands available under the `/vis~/refresh/` command subdirectory are:

Command	Arguments	Guidance
`/vis~/refresh/view`		Optimal redraw of current view of current scene - uses double buffer and graphical database, if any.

The commands available under the `/vis~/set/` command subdirectory are:

Command	Arguments	Guidance
/vis~/set/culling	true/false	Global culling flag. Does not change specific culling flags.
/vis~/set/cull_covered_daughters	true/false	Cull (i.e., do not draw) daughters covered by opaque mothers.
/vis~/set/cull_invisible_objects	true/false	Cull (i.e., do not draw) “invisible” objects.
/vis~/set/cull_by_density	choice, density	Cull (i.e., do not draw) geometry objects with density less than specified — on/off and density (g / cm3).
/vis~/set/drawing_style	choice	Style of drawing (wireframe, etc.).
/vis~/set/marker_choices	choice	Choices for marker drawing (hidden by surfaces, etc.).
/vis~/set/rep_style	choice	Style of graphics representation for geometrical volumes (polyhedron, NURBS,...).
/vis~/set/scene	choice	Several ways of setting the scene. For example, choosing a physical volume.
/vis~/set/section	choice, plane	Set plane for drawing section (DCUT). Specify plane by x y z units nx ny nz, e.g., for a y-z plane at x = 1 cm: /vis~/set/section_plane on 1 0 0 cm 1 0 0
/vis~/set/sides	n	Number of sides per circle in polygon approximation.
/vis~/set/verbose	integer	Controls amount of printing (0 = quiet).
/vis~/set/view	choice	Make this view current.

The commands available under the /vis~/show/ command subdirectory are:

Command	Arguments	Guidance
/vis~/show/view		Make the view visible, if not so already (initiates post-processing for graphics systems which use such techniques).

The commands available under the /vis~/expert/ command subdirectory are:

Command	Arguments	Guidance
/vis~/expert/draw_circle	x, y, z, *size*	Specify coordinates and world size.
/vis~/expert/draw_line	x_0, y_0, z_0, x_1, y_1, z_1	Specify coordinates of start and end.
/vis~/expert/draw_marks_and_show	x, y, z, *size*	Specify coordinates and world size.
/vis~/expert/draw_physical_volume		A test of `G4VisManager::Draw (const G4VPhysicalVolume&,...`
/vis~/expert/draw_polymarkers	N_{spirals}	Draws `G4Polymarkers` in the form of coloured spirals.
/vis~/expert/draw_spiral		Draws a `G4Polyline` in the form of a spiral.
/vis~/expert/draw_spirals	N_{spirals}	Draws `G4Polylines` in the form of coloured spirals.
/vis~/expert/draw_square	x, y, z, *size*	Specify coordinates and world size.

C Visualization Attributes

Many different types of GEANT4 object may have Visualization Attributes, or, rather, they may have a *pointer* to a `const G4VisAttributes` object. The GEANT4 objects aggregate a pointer, rather than a Visualization Attributes object itself, in order to keep overheads down. This places the burden of instantiating and maintaining the Visualization Attributes objects on the user. In particular, he must ensure that the life of the Visualization Attributes object is at least as long as the object being drawn. Usually, this is no problem, since the Visualization Attributes object is usually created at the same time as the object being drawn, so the same method of instantiation — automatic (local) or with the `new` operator — should be used for both. (In future releases there may be a Visualization Attributes Manager and a Visualization Attributes Store to assist the user.)

C.1 Available Visualization Attributes

```
G4bool      fVisible;            // Visibility flag
G4Colour    fColour;
LineStyle   fLineStyle;
G4double    fLineWidth;          // Units of "normal" device linewidth, e.g.,
                                 // pixels for screen, 0.1 mm for paper.
G4bool      fForceDrawingStyle;  // To override view parameters.
ForcedDrawingStyle fForcedStyle;
```

C.2 Setting Visualization Attributes

Here are the available constructors and `Set` methods:

```
enum LineStyle {unbroken, dashed, dotted};
enum ForcedDrawingStyle {wireframe, solid};

G4VisAttributes ();
G4VisAttributes (G4bool visibility);
G4VisAttributes (const G4Colour& colour);
G4VisAttributes (G4bool visibility, const G4Colour& colour);

static const G4VisAttributes Invisible;


void SetVisibility     (G4bool v);
void SetColour         (const G4Colour& colour);
void SetColor          (const G4Color&  color);
void SetColour         (G4double red, G4double green, G4double blue,
                        G4double alpha = 1.);
void SetColor          (G4double red, G4double green, G4double blue,
                        G4double alpha = 1.);
void SetLineStyle      (LineStyle style);
void SetLineWidth      (G4double w);
void SetForceWireframe (G4bool force);
void SetForceSolid     (G4bool force);
```

Here is an example of how a Visualization Attributes object with chosen Visualization Attributes can be made.

```
square.SetWorldSize (worldSize);
G4Colour c (1.0, 0.0, 0.0 );
G4VisAttributes a(c);
square.SetVisAttributes(&a);
```

Here is an example of how the Visualization Attributes of an existing GEANT4 object can be changed or attributed to another GEANT4 object.

```
// pPhysVol is the pointer to the physical volume.
// transform is the required G4Transform3D in the world
//   coordinate system.

// Find the corresponding logical volume...
G4LogicalVolume* pLV  = pPhysVol -> GetLogicalVolume ();

// Copy its visualization attributes into local variable...
const G4VisAttributes* pVA = pLV -> GetVisAttributes ();
G4VisAttributes attribs;     // Default vis attributes.
if (pVA) attribs = *pVA;     // Set to those of logical volume if any.

// Modify the visualization attributes...
G4double red   = G4UniformRand();
G4double green = G4UniformRand();
G4double blue  = G4UniformRand();
G4Colour c (red, green, blue);
attribs.SetColour (c);
attribs.SetForceSolid (true);

// Draw it...
fpVMan -> Draw (*pPhysVol, attribs, transform);
```

Don't forget, the Visualization Attributes must have a life at least as long as the object to which they are attributed.

D Drawing in User Action Functions

In any of the user action functions the user may write code to draw something. However, the excutable that he using may or may not include code to instantiate the Visualization Manager so he must protect all drawing code as follows:

```
G4VVisManager* pVisManager = G4VVisManager::GetConcreteInstance()
if (pVisManager) {
  ... drawing code
}
```

The function `G4VVisManager::GetConcreteInstance()` returns a null pointer if the Visualization Manager has not been instantiated.

D.1 The C++ Abstract Interface of the Visualization Manager

`G4VisManager` is actually derived from the abstract base classs `G4VVisManager`, and it is the latter that defines what is available to the user in user action routines.

Note that the `G4Transform3D` objects refer to the transformation of the *object* being drawn. However, for some functions, there is a version which takes a `G4Translation` and a `G4RotationMatrix`, the latter being a *system* rotation, as in `G4PVPlacement`.

Note also that where a `G4VisAttributes` argument is specified, it overrides any attributes belonging to the object itself. Otherwise, the visualization attributes are assumed to be those belonging to the object being drawn (you can set its attributes — see Section C.2).

```
virtual void Draw (const G4Polyline&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4Text&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4Circle&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4Square&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4Polymarker&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4Polyhedron&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4NURBS&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4VSolid&, const G4VisAttributes&,
```

```
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4LogicalVolume&, const G4VisAttributes&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;

virtual void Draw (const G4VPhysicalVolume&, const G4VisAttributes&,
  const G4Transform3D& objectTransformation = G4Transform3D::Identity) = 0;
```

D.2 Drawing a Particle Step

Here is an example from `G4UserSteppingAction.cc`:

```
G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
static int coutCount = 0;
if (coutCount < 10) {
  coutCount++;
  cout << "MySteppingAction::UserSteppingAction called." << endl;
}
if(pVVisManager) {
  G4Polyline pl;
  G4Colour c;
  G4Step* pStep = GetSteppingManager () -> GetStep ();
  G4double chg =
    pStep -> GetTrack () -> GetDefinition () -> GetPDGCharge ();
  if (chg != 0.) c = G4Colour (1., 0., 0.);
  else           c = G4Colour (0., 1., 0.);
  G4VisAttributes va (c);
  pl.SetVisAttributes (&va);
  pl.append (pStep -> GetPreStepPoint () -> GetPosition ());
  pl.append (pStep -> GetPostStepPoint () -> GetPosition ());
  static int coutCount = 0;
  if (coutCount < 10) {
    coutCount++;
    cout << "MySteppingAction::UserSteppingAction: drawing "
         <<  pl << endl;
  }
  pVVisManager -> Draw (pl);
}
```

Don't forget to test the pointer to G4VVisManager — it is null (false) if the G4VisManager has not been instantiated.

D.3 Drawing a Physical Volume

The user may wish to draw a particular physical volume in a particular colour to represent a hit. For example, the colour might represent the energy deposited in a calorimeter element. This is how it's done.

```
// pPhysVol is the pointer to the physical volume.
// transform is the required G4Transform3D in the world
//   coordinate system.
```

```
// Find the corresponding logical volume...
G4LogicalVolume* pLV  = pPhysVol -> GetLogicalVolume ();

// Copy its visualization attributes into local variable...
const G4VisAttributes* pVA = pLV -> GetVisAttributes ();
G4VisAttributes attribs;     // Default vis attributes.
if (pVA) attribs = *pVA;     // Set to those of logical volume if any.

// Modify the visualization attributes...
G4double red   = G4UniformRand();
G4double green = G4UniformRand();
G4double blue  = G4UniformRand();
G4Colour c (red, green, blue);
attribs.SetColour (c);
attribs.SetForceSolid (true);

// Draw it...
fpVMan -> Draw (*pPhysVol, attribs, transform);
```

D.4 Event Timing Messages

The Event Manager should keep the Visualization Manager informed about the status of event processing:

E Useful Quantities From View Parameters

These functions encapsulate the calculation of useful quantities derivable from the View Parameters.

```
// Here follow functions to evaluate the above algorithms as a
// function of the radius of the Bounding Sphere of the object being
// viewed.  Call them in the order given - for efficiency, later
// functions depend on the results of earlier ones (store the
// results of earlier functions in your own temporary variables -
// see, for example, G4OpenGLView::SetView ().)

G4double G4ViewParameters::GetCameraDistance (G4double radius) const {
  G4double cameraDistance;
  if (fFieldHalfAngle == 0.) {
    cameraDistance = radius;
  }
  else {
    cameraDistance = radius / sin (fFieldHalfAngle) - fDolly;
  }
  return cameraDistance;
}

G4double G4ViewParameters::GetNearDistance (G4double cameraDistance,
                                            G4double radius) const {
  const G4double small = 1.e-6 * radius;
```

```
  G4double nearDistance = cameraDistance - radius;
  if (nearDistance < small) nearDistance = small;
  return nearDistance;
}

G4double G4ViewParameters::GetFarDistance (G4double cameraDistance,
                                           G4double nearDistance,
                                           G4double radius) const {
  G4double farDistance = cameraDistance + radius;
  if (farDistance < nearDistance) farDistance = nearDistance;
  return farDistance;
}

G4double G4ViewParameters::GetFrontHalfHeight (G4double nearDistance,
                                               G4double radius) const {
  G4double frontHalfHeight;
  if (fFieldHalfAngle == 0.) {
    frontHalfHeight = radius / fZoomFactor;
  }
  else {
    frontHalfHeight = nearDistance * tan (fFieldHalfAngle) / fZoomFactor;
  }
  return frontHalfHeight;
}
```

F The C++ Abstract Interface of the Scene

"Scene" functions are not usually called directly by the user. They are invoked by GEANT4 geometrical solids when the user asks for it to be drawn, as, for example, when drawing a physical volume in Appendix C.2, or by asking for the detector to be drawn at the User Interface. `G4VScene` is itself derived from `G4VGraphicsScene`, and it is the latter that defines what is available to the GEANT4 kernel.

```
class G4VGraphicsScene {

public:

  ///////////////////////////////////////////////////////////////////
  // Functions for adding raw GEANT4 objects.  The concrete graphics
  // scene has the option of implementing its own model or asking the
  // solid to provide a G4Polyhedron or similar primitive - see, for
  // example, G4VScene in the Visualization Category.

  virtual void AddThis (const G4Box&    box)    = 0;
  virtual void AddThis (const G4Cons&   cons)   = 0;
  virtual void AddThis (const G4Tubs&   tubs)   = 0;
  virtual void AddThis (const G4Trd&    trd)    = 0;
  virtual void AddThis (const G4Trap&   trap)   = 0;
  virtual void AddThis (const G4Sphere& sphere) = 0;
  virtual void AddThis (const G4Para&   para  ) = 0;
  virtual void AddThis (const G4Torus&  torus ) = 0;
  virtual void AddThis (const G4VSolid& solid)  = 0;  // For solids not above.

  ///////////////////////////////////////////////////////////////////
```

```
  // Other functions.

  virtual void EstablishSpecials (G4PhysicalVolumeModel&) {}
  // Used to establish any special relationships between scene and
  // this particular type of model - non-pure, i.e., no requirement to
  // implement.  See G4PhysicalVolumeModel.hh for details.

  virtual void DecommissionSpecials (G4PhysicalVolumeModel&) {}
  // Used to reverse the effect of EstablishSpecials, if required.

  virtual void PreAddThis (const G4Transform3D& objectTransformation,
                           const G4VisAttributes& visAttribs) = 0;
  // objectTransformation is the transformation in the world
  // coordinate system of the object about to be added, and
  // visAttribs is its visualization attributes.

  virtual void PostAddThis () = 0;

};
```

References

[1] DAWN (Drawing for Academic WritiNgs) or Fukui Renderer. Available with the GEANT4 distribution or by anonymous ftp via `ftp://i1nws2.fuis.fukui-u.ac.jp/pub/graphics/fukui_graphics_system/`. M. Kawaguti, S. Tanaka, "3D Graphics Module for Modeling, Visualization and Analysis", Fukui University preprint, FU-IS/HEP-95-01, Fukui University, 9-1, Bunkyo 3-chome, Fukui, 910 Japan, and paper presented at CHEP97.

[2] OPACS ("O" Packages), Guy Barrand, LAL, Orsay. See HEPVIS95 and `http://www.lal.in2p3.fr/opacs/The_o_packages.html` and a paper presented at CHEP97.

[3] Mesa, a client-side-implementation of OpenGL, via anonymous ftp from `iris.ssec.wisc.edu` in `pub/Mesa`. Being client-side-implemented has the advantage that it runs over the X protocol without the need for a GLX server.

[4] Contact the CERN User Computer Office for details. At the time of writing, CERN has purchased a number of OpenGL and Open Inventor developers licences and a larger number of run-time licences for the CERN site. The large collaborations, through CERN, are currently negotiating for outside institutes.

[5] The GEANT4 Visualization Manager and Graphics System Interface Specification and Reference, to be published.