Graphics and Fast Algorithms for Particle Physics Detector Simulations.

Andrew Walkden

THE UNIVERSITY *of* MANCHESTER

High Energy Group
Department of Physics and Astronomy

A thesis submitted to The University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering.

May 1999

Contents

List of Figures

List of Tables

Abstract

Geant4 is the first global collaborative effort in high energy physics to employ object oriented technologies in pursuit of a single goal. The portion of this effort devoted to providing OpenGL drivers written in object oriented C++ is presented here, together with a discussion of the framework that Geant4 provides in favour of visualization. Also described is the method by which physics processes may become parameterized by Geant4, and an investigation into a possible parameterization of e^+, e^-, and γ initiated electromagnetic showers in Caesium Iodide doped with Thallium, that might be used in the BaBar experiment.

Declaration

The author

The author was educated at the Manchester Grammar School between 1984 and 1991 and the University of Manchester between 1992 and 1995, gaining an upper second class honours degree in Physics, before joining the High Energy Physics Group at the University of Manchester in 1995. The work presented here was undertaken in Manchester and funded by a Samuel Gratrix studentship.

Chapter 1

Detector simulation for particle physics

1.1 Introduction to Geant4

The role of the physicist is to observe, record and try to see the patterns in the surrounding world. When a pattern is found, the physicist may try to describe it and, based on this description, make predictions about the quantities that constitute the pattern. In modern particle physics, machines are used to observe and record data, and computers are employed to help store, process and visualize that data. Computer programs are also able to simulate and visualize the processes observed by the machines, using descriptions formed from previous inspections of data. Geant4 is one such computer program.

Geant4 has been developed as a research and design project (RD44) at CERN[1]. It aims to provide an object oriented toolkit of modules, written in C++, regarding the simulation and visualization of physics processes of particles in matter. The toolkit philosophy of production will enable users to supply their own modules to

[1]The European particle physics laboratory.

the Geant4 framework, relieving some of the burden of software maintenance from the originators of code. Features of object orientation with C++ (chapter 2) and visualization with OpenGL (chapter 3) in Geant4 (chapter 5) are described, together with a description of the parameterization framework in Geant4 (chapter 6). Also described is an investigation into a parameterization of electromagnetic showers in Caesium Iodide doped with Thallium (chapter 7), which might be inserted into this framework for use in a simulation of the BaBar experiment at SLAC[2]. The BaBar experiment is due to start taking data in early 1999 and will be the first particle physics experiment to use the Geant4 toolkit. Use of the toolkit is not, however, limited to particle physics, as applications in medical physics, space physics and other areas of physics are being evaluated.

1.2 Introduction to BaBar

The BaBar experiment is situated at the interaction vertex of the asymmetric 9.0 GeV e^- on 3.1 GeV e^+ PEP-II storage ring facility at SLAC. This places the centre of mass energy on the $\Upsilon(4S)$ resonance, whose dominant decay mode is to $B\bar{B}$. The asymmetry of the system boosts the centre of mass in the laboratory frame enabling the short lived ($\tau \sim 10^{-12}$ s) B mesons and their decay chains to be observed.

In measuring the decays of B^0 and $\bar{B}^0$ mesons, it is anticipated that the effects of CP violation will be evident, as were first observed in the decay of K^0_L mesons ($\tau \sim 10^{-8}$ s) in 1964 by Cronin and Fitch. Although the $B\bar{B}$ production rate is only going to be 3 Hz, rising to 10 Hz at the final beam luminosity of $10^{34}\,cm^{-2}\,s^{-1}$, the boosted centre of mass and ensuing separation of the B and $\bar{B}$ vertices promises to provide enough $B^0\bar{B}^0$ decay information (including the time between the decay of the B^0 and $\bar{B}^0$) to help make accurate determinations of some elements of the Cabibbo-Kobayashi-Maskawa quark mixing matrix.

[2]Stanford Linear Accelerator Center, San Francisco, USA.

1.3 The BaBar electromagnetic calorimeter

Electromagnetic calorimetry for the BaBar experiment will be performed by a barrel calorimeter surrounding the interaction vertex and covering a solid angle range of $-0.80 \leq \cos\theta \leq 0.89$, and an endcap calorimeter covering the solid angle range $0.89 \leq \cos\theta \leq 0.97$. The calorimeters are made from trapezoidal crystals of Caesium Iodide doped with Thallium (CsI(Tl)), with 5880 such crystals comprising the barrel, and 900 comprising the endcap. Each crystal is between 16.0 and 17.5 radiation lengths deep, and is backed by two photodiodes (two for redundancy) and the readout cables to an ADC[3] board, as well as a fibre optic cable present for in-situ calibration purposes.

By its very nature (the electromagnetic calorimeter is designed to absorb and measure as much energy via electromagnetic interactions as possible) the simulation of interactions undergone by charged particles traversing the electromagnetic calorimeter can be a very lengthy process, and a prime candidate for parameterization.

[3]Analogue to Digital Converter.

Chapter 2

Object Oriented C++

2.1 Overview of object oriented programming

Object oriented programming (OOP) is a style of computer programming that emphasizes the association between data and their behaviour. It is a development of programming style, that surpasses more traditional procedural (structured) formalisms, designed with larger, multi-developer projects in mind [16]. Programs can be written in an object oriented style in any language but some languages, such as Smalltalk, Eiffel [12], C++ [17], [5], and more recently Java, lend themselves to it. As a program grows in size, so does its complexity. As the complexity increases, the architecture of the program becomes more significant than the language it is made from. The employment of OOP can help to minimize the impact of these increases in complexity, for example, by making behaviour independent of the underlying implementation (see section 2.2.2).

To help achieve a good object oriented program structure, several good Computer Aided Software Engineering (CASE) tools exist on the market. One such CASE tool, Rational Rose, was employed to help produce some of the diagrams in this report. A brief description of the UML[1] is given in Appendix A.

The GEANT4 toolkit was designed in an object oriented fashion, and has used C++ to help achieve the design goals. Some salient features of OOP and how they are realized in C++ are discussed below.

2.2 Object orientation

The mainstay of any object oriented language is the object. An object is a collection of data and functions (or methods) that together define a single entity, which in C++ is called the class. The class may then be used as a blueprint to create many 'instances' of itself, referred to as objects. The user defined object is treated on equal terms with objects of intrinsic data types (float, int, char, etc.) and so augments the language. The real power of an object oriented program is in the quality of design of its types.

However, C++ merely empowers the programmer to create an OO project; it does not enforce it. Consequently, a lot of care has to be taken in ensuring that a large project such as Geant4 does not assume that the use of C++ guarantees OOP. The goals of reuse, quality, and maintainability are not automatically achieved in C++.

The attractions of basing a programming effort on the design and creation of objects are numerous. Three of the more important reasons are discussed below, with examples of their expression in C++ and Geant4.

2.2.1 Inheritance

Inheritance is the mechanism by which OOP aims to achieve code sharing and reuse. It is expressed by the derivation of one class from another — for instance

[1]Universal Modeling Language.

`G4OpenGLXmView` is derived from `G4OpenGLXView` in figure 2.5. A derived class may build on the parent class by reusing all of the parent's functionality, or it may redefine parts or all of the implementation. If it is to redefine functions then the functions must be declared as being `virtual` in the parent class so that they may be added to the so-called virtual function table in preparation for dynamic binding at run time. If the parent class declares a function as being *pure* `virtual` then this function *must* be defined in any derived class or else that class will be abstract (unable to be instantiated).

```
virtual void Draw () = 0;
```

Figure 2.1: The declaration of a pure virtual function

Another consequence of inheritance is that an object of a derived class type may be referred to as an object of the parent class type and treated as such where it may be convenient to do so. (See example in 2.2.3.)

2.2.2 Encapsulation

The concept of encapsulation describes the distinction between an object's internal representation and its external interface. In C++ there are three levels of visibility: `public`, `protected`, and `private`. These enable a class to limit the extent to which objects of other classes may interact with objects of its own type. Members (data or functions) of a class may also be declared as `const`, `static`, or `extern`. The meanings of these keywords are described below.

`public:`
the `public` keyword signifies that a member (datum or function) of a class can be accessed from any other class or function (i.e. anywhere). The constructor function of a class would normally be made `public`.

`protected:`
a `protected` member may be accessed from an instance of the class or any of its derived classes but nowhere else. `protected` is a way of keeping access 'in the family' for a class hierarchy.

`private:`
a `private` member can only be accessed from the instance of its own class.

`const:`
a `const` data member must be defined at the same time as its declaration, e.g. `const double pi = 3.14159;` as it may not be modified after declaration. A compile time error is caused if an attempt is made to do so. In the context of functions, `const` may occur in two places. When positioned in front of the function name, `const` acts on the return type of the function so that the value returned by successful execution of the function may not be changed. When positioned after the function name, `const` signifies that no data members of the object may become altered during the execution of the function.

```
G4bool GetValue () const;
const G4VisExtent& GetBoundingSphereExtent ();
const G4ViewParameters& GetViewParameters () const;
```

Figure 2.2: Some uses of `const` with functions in Geant4. (i) a `const` function, (ii) a function with a `const` return type, (iii) a `const` function with a `const` return type.

`static:`
data members of a class that are declared as `static` are shared by all instances of that class, stored uniquely in one place, and can be accessed independent of an instance of the class — thus : *class_name*::*data_member*. A `static` function of

a class is designated by the `static` keyword appearing in the function declaration before the function name and means that it can be accessed independently of any instance of the class. Also, it has no access to any non `static` data members of that class.

`extern`:
the `extern` keyword in C++ allows data and functions to have global scope. Any data or functions that are declared outside a function are automatically made `extern`. This becomes useful when a global variable is defined in one file and is required in another. By declaring the variable as `extern` in the file where it is required, the linker is told to resolve the reference to the variable by looking for its definition elsewhere.

in file1.cc

```
double pi = 3.14159;
```

in file2.cc

```
extern double pi; // tells compiler that pi is declared elsewhere

double Circumference (Circle c) {
return 2.0*pi*c.radius();
}
```

Figure 2.3: The `extern` storage class in C++

The declaration of object members as `public`, `protected`, or `private` should be viewed as another aid to code reuse. These keywords can help avoid the innocent mistakes that users unfamiliar with the code they are trying to reuse might conceivably make. Such declarations provide protection against a cluttered global variable space such as that offered by COMMON blocks in FORTRAN. As with any language that gives direct access to raw memory, there is no protection against fraudulent use.

2.2.3 Polymorphism

Polymorphism describes a situation in which a single name (e.g. of a function) can represent many different implementations. Object orientation allows polymorphism to be expressed in three different ways.

(i) Function overloading

In C++, function overloading is achieved via a process of ‘name mangling’. The compiled name of each function is constructed to include information on the number and types of arguments with which the function would be called. This enables the linker to select the appropriate overloaded function, given the context.

```
void G4OpenGLScene::AddPrimitive (const G4Text& text)
void G4OpenGLScene::AddPrimitive (const G4Circle& circle)
void G4OpenGLScene::AddPrimitive (const G4Polyhedron& polyhedron)
```

Figure 2.4: Some polymorphic overloaded functions in Geant4 visualization

(ii) Polymorphism via inheritance

Where a class hierarchy exists, functions supplied by a parent class can also be defined in a derived class. If a particular function in this category is declared as `virtual`, then its invocation is only decided upon at run time, i.e. the function becomes *dynamically bound*. This action lends itself well to code reuse as the same piece of code can lead to different consequences depending on which objects in a hierarchy have been instantiated.

With reference to figure 2.5, the `virtual` function mechanism allows the correct form of `CreateMainWindow()` to be called in the statement

`G4OpenGLXView* v = new G4OpenGLXmView(); v->CreateMainWindow();`.

```
class G4OpenGLXView: virtual public G4OpenGLView {
protected:
virtual void CreateMainWindow ();
...
}
class G4OpenGLXmView: public G4OpenGLXView {
protected:
virtual void CreateMainWindow ();
...
}
```

Figure 2.5: Polymorphism via inheritance in Geant4

(iii) Polymorphism via genericity

C++ gives support for parameterized forms of classes via the `template` keyword. Template classes can save a lot of class rewriting when the only differences between classes would be type declarations. They are particularly useful as containers or other such objects that may perform operations (on a group of data) which may be generalized so as to be independent of the data type involved.

Because template classes are compiled they benefit from the C++ compiler's strong type checking. (Similar results may be achieved through the use of precompiler macro definitions but these bypass the type checking.)

The standard template library is a collection of templates (mostly containers and iterators) that makes a large contribution to code reuse. These have been tested and may be assumed to function correctly although their introduction into projects (such as Geant4) in place of the existing template classes is not trivial.

Chapter 3

The OpenGL Graphics Library

The OpenGL application programming interface (API) is an open standard for the production of 3D graphics. It originated as the proprietary Graphics Library (GL) for Silicon Graphics' IRIS systems and has been adopted as a *de facto* standard for the production of 2D and 3D computer graphics. OpenGL is a procedural library so every detail of a view (camera position, lights, object positions, etc.) must be described before it can be rendered [11].

The term 'open' means that the language definition is free from licensing restrictions, enabling free-ware versions, such as Brian Paul's Mesa [14], to be written. It defines over 120 functions that perform a fairly basic set of 3D graphics operations, such as rotate, translate, scale, etc., as well as providing some functionality in respect of more sophisticated 3D graphics operations such as texture mapping or evaluated object support. Some essential features of 3D graphics programming are discussed below, together with some further aspects of OpenGL.

3.1 Fundamentals of 3D graphics

There are three operations at the very heart of all 3D graphics modeling: scale (S), rotate (R), and translate (T). If a point P in euclidean three dimensional space is represented as a 1×3 column vector then these operations may be represented by a 1×3 matrix (translation) and two 3×3 matrices. The operation of translation is additive whereas scaling and rotation are multiplicative.

Translation : $P' = P + T$

Rotation : $P' = R \cdot P$

Scaling : $P' = S \cdot P$

3.1.1 The homogeneous coordinate system

If we choose to represent the points in three dimensional euclidean space as four dimensional *homogeneous coordinates* then translation may also be treated as a multiplicative process, enabling us to apply translations, scalings, and rotations in a consistent fashion. Homogeneous coordinates were introduced to geometry in 1946 by E. A. Maxwell [10] and are constructed from the x, y and z components of euclidean space, plus a fourth, W component. To get from euclidean to homogeneous coordinates one may simply add the W component set to 1. Given a point in homogeneous coordinate space, all multiples of it (i.e., points lying on the line joining it with the origin) refer to the same point in euclidean space. Furthermore, the plane in homogeneous coordinate space with $W = 1$ represents the complete set of points in euclidean space.

The 4×4 matrices T, R, and S, used to produce translation, rotation (about the $z - axis$), and scaling respectively may be represented as follows.

$$T = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R = \begin{pmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3.1.2 Compound operations

When combining S, T, and R into a compound matrix, caution must be exercised in the order of operations. T commutes with neither R nor S and so the order in which these operations are compounded matters. Assuming that a correct 4×4 compound matrix (M) has been formed, the premultiplication of a 1×4 homogeneous coordinate vector, H, by it would appear to involve 16 multiplications and 12 additions. However, the bottom row of M is always 0 0 0 1 (for a homogenized matrix — one in which all elements have been divided by W) so the operation reduces to 9 multiplications and 9 additions.

3.2 Stages of modeling in OpenGL

Before OpenGL can draw anything, a model must be specified. A model is a collection of vertices (or an evaluated object(s) such as a NURBS, from which a set of vertices may be derived) with which other data such as colour may be associated. The model must eventually become clipped (to discard vertices outside some region of interest) and represented in 2D to enable visualization in some *viewport* (on screen, paper, etc.). The processes employed between the construction of the model and its visualization are described briefly below.

3.2.1 The modelview transformations

In constructing the model, rotations, translations, and scalings may be combined and applied as *modeling transformations* or *viewing transformations.* The term *viewing transformation* means a transformation applied to the viewer (camera) and the term *modeling transformation* means a transformation applied within (or to) the model. The viewer in OpenGL is initially located at the origin and faces down the negative $z - axis$. Moving the model ten units in the negative z direction is equivalent to moving the viewer ten units in the positive z direction. This equivalence makes it sensible to think about 'modelview' transformations rather than about modeling and viewing ones separately.

The application of modelview transformations in OpenGL takes place via the modelview matrix stack. Each time a modelview transformation is specified, it postmultiplies the top matrix in the stack to produce a new transformation matrix. This matrix becomes the new top modelview stack matrix. There may be over 32 matrices in the modelview matrix stack (depending on the OpenGL implementation). This enables the programmer to 'push' and 'pop' the top matrix if there is reason to preserve a copy of its contents. The action of pushing the stack moves all matrices in the stack down one position (and so may discard the contents of what was the 32nd matrix before the push) and copies the contents of the 2nd place matrix into the new top matrix. The programmer may restore the original top matrix at any time by popping the matrix stack, which moves all the matrices up one position (discarding the contents of the top matrix).

Each vertex described to OpenGL becomes premultiplied by the current top modelview matrix. Hence, the last transformation described to OpenGL is the first transformation to be applied to a vertex. In this way, the order and effect of the application of modelview transformations in OpenGL on a point P is equivalent to the interpretation of transformations being applied to the vertex (rather than to the camera).

OpenGL code

```
glRotatef (...);
glTranslatef (...);
glScalef (...);

glVertex3f (...);
```

$\longrightarrow$

Implied matrix operations in OpenGL

$M = I$

$M = R$

$M' = M \times T$

$M'' = M' \times S$

$P' = M'' \times P$

$= (R \times (T \times (S \times P)));$

3.2.2 The projection transformations

Once the model has been composed, it must be clipped in order to discard any vertices lying outside a particular *viewing volume*. The points lying within the viewing volume must also become projected onto a 2D plane before they can be mapped into the viewport. The tasks of clipping and projection onto 2D planes are performed by the *projection matrix*. OpenGL actually produces projection matrices which project vertices into a normalized device coordinate system (NDCS) box between (-1, -1, -1) and (1, 1, 1).

Although only x and y positions are needed to map the projection into a viewport, the depth information is retained so that, if requested, rendering does not obscure foreground pixels with background ones. Two kinds of projection matrix can be defined in OpenGL; orthographic and perspective.

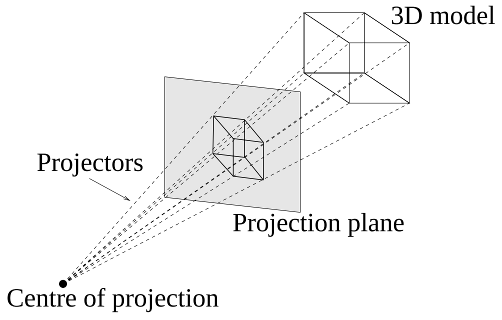

Figure 3.1: The projection of a 3D model onto a 2D projection plane

The orthographic projection matrix

A view of a model projected with an orthographic projection matrix preserves parallelism of lines in the model but not angles. For this reason, orthographic projection is useful for technical drawing but lacks realism, as foreshortening of lines is uniform rather than dependent on the distance from the centre of projection (as is the case with the perspective projection).

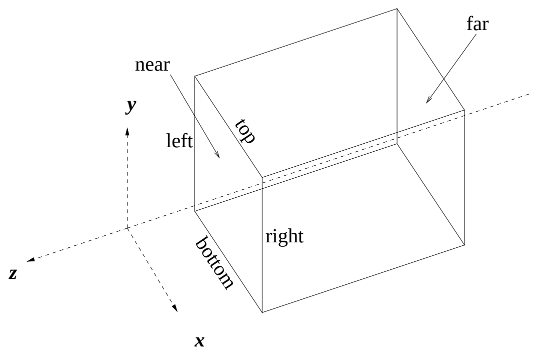

Figure 3.2: The orthographic projection viewing volume

To construct the orthographic projection matrix, one considers its role of mapping coordinates in the modeling coordinate system (MCS) into a NDCS box between (-1, -1, -1) and (1, 1, 1).

From figure 3.3, $x_{NDCS} = \frac{2x_{MCS}}{(right-left)} - \frac{(right+left)}{(right-left)}$. Similarly,

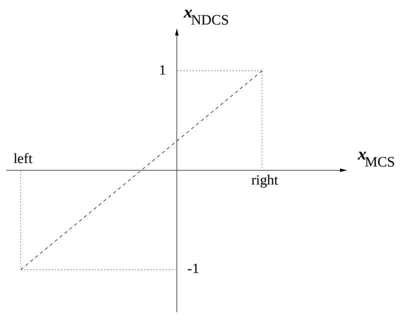

Figure 3.3: Mapping MCS to NDCS in an orthographic projection

$y_{NDCS} = \frac{2y_{MCS}}{(top-bottom)} - \frac{(top+bottom)}{(top-bottom)}$

$z_{NDCS} = -\frac{2z_{MCS}}{(far-near)} - \frac{(far+near)}{(far-near)}$

In a 4×4 matrix, this is represented as

$$M_{ortho} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\left(\frac{r+l}{r-l}\right) \\ 0 & \frac{2}{t-b} & 0 & -\left(\frac{t+b}{t-b}\right) \\ 0 & 0 & -\left(\frac{2}{f-n}\right) & -\left(\frac{f+n}{f-n}\right) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The perspective projection matrix

A perspective projection is characterized by parallel lines in the model, which are not parallel to the plane of projection, converging to a *vanishing point* in the view. It is more realistic than the orthographic projection, as the size of a model element under a perspective projection is inversely proportional to the distance of that element from the centre of projection. This is called perspective foreshortening.

The production of the matrix for a perspective projection is rather less straightforward than for the orthographic case. In projecting points from the frustum viewing volume of MCS space to a NDCS space, each of x, y, and z must be scaled and translated. Furthermore, x_{NDCS}, and y_{NDCS} must become functions of z_{MCS}. The

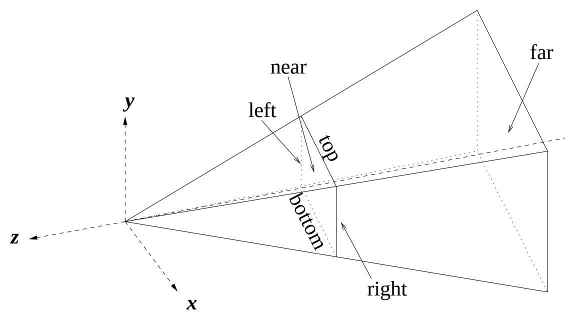

Figure 3.4: The perspective projection viewing volume

matrix below is able to accommodate these requirements.

$$M_{persp} = \begin{pmatrix} A & 0 & B & 0 \\ 0 & C & D & 0 \\ 0 & 0 & E & F \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

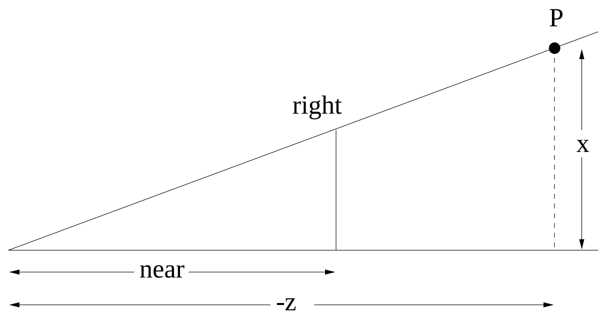

Figure 3.5: Considerations for constructing matrix elements of M_{persp}

To calculate the matrix elements, one inspects the purpose they must serve. We require that for $x_{MCS} = -z_{MCS} \times \frac{right}{near}$, $\frac{x_{NDCS}}{W_{NDCS}} = 1$.
The transformed x component in NDCS space is $x_{NDCS} = A \times x_{MCS} + B \times z_{MCS}$.

Noting that $W_{NDCS} = -z_{MCS}$, we get $\frac{A\times(-z_{MCS}\times\frac{right}{near})+B\times z_{MCS}}{-z_{MCS}} = 1$.

Similarly, $\frac{A\times(-z_{MCS}\times\frac{left}{near})+B\times z_{MCS}}{-z_{MCS}} = -1$.

The last two can be solved for A and B to give

$A = \frac{2 \times near}{right - left}$, and $B = \frac{right + left}{right - left}$.

The other elements of M_{persp} are produced in the same way to give

$$M_{persp} = \begin{pmatrix} \frac{2 \times n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2 \times n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

3.2.3 The viewport transformation

When the model has been transformed into NDCS space, it is ready to be mapped onto the screen (or file, paper, etc.). OpenGL supplies the function `glViewport(x, y, width, height)` to define the pixel dimensions of the window in which the model is to be *rasterized* or converted into a pixel representation. It is at the rasterization stage that information given by the z_{NDCS} coordinates may be used to apply depth buffering or fog effects to the view of the model.

3.3 Colour

The processes discussed so far can produce an image on a 2D screen of a 3D computer model but no mention has yet been made of how to propagate colours. OpenGL is a state machine, meaning that its current state will persist without the need for redefinition, until some signal is received that causes its state to change. More simply, each time a vertex is specified in the model, the current colour state is associated with it (which initially defaults to white). Thus, OpenGL's current colour state must be changed — via the `glColor*(...)` function — before specification of every vertex that is intended to be a different colour from the last one. During the rasterization process, OpenGL decides which pixels in the viewport each of the primitives (i.e., polygon, line, point, bitmap, or image) that comprise the model occupies. Each pixel then accumulates a value for z (depth), colour, and texture coordinates (if texture mapping is enabled) from the part of the primitive it is meant to represent. This collection is referred to as a *fragment*.

3.3.1 Colour in an unlit model

Each primitive for which fragments are produced is defined by a set of vertices. In an unlit view, the colour associated with the fragment may be simply derived from the colour associated with its primitive's vertices. If the vertices of the primitive are not all the same colour, then the colour of the fragment is derived from a smooth interpolation between the colours of the vertices.

3.3.2 The illuminated model

OpenGL also offers the opportunity to specify a number of lights (up to at least 8) with which to illuminate the model. As soon as light is introduced into the model, surface properties of primitives becomes important.

Lights

The properties and positions of the lights that may illuminate the model are specified to OpenGL by the `glLight*(...)` function. This may define the position of each light in the model as well as its *specular*, *diffuse*, and *ambient* red, green, blue, and alpha (RGBA) colour components. The alpha component may be used when calculating the colour components of fragments where model primitives overlap, and is best thought of as opacity. Hence, an alpha value of 1.0 represents a totally opaque colour, and alpha values of less than 1.0 imply an increasing degree of transparency. Properties of spot lights (spread angle, intensity distribution, direction, etc.) and light attenuation coefficients (constant, linear, and quadratic) may also be specified.

- Specularity

Specular light has a definite direction and scatters off objects in a preferred direction. An example of specular light is sun light reflecting off a highly polished mirror.

- Diffuseness

Diffuse light comes from a definite direction, so is more intense in reflection if its incidence with an object is 'square on' rather than glancing, but is equally likely to scatter in all directions when reflecting off an object.

- Ambience

Some component of light from a source is scattered so much by the environment that its direction is impossible to determine. When ambient light reflects off a surface, it does so in all directions.

Materials

In OpenGL, one may specify terms to describe how the various components of incident light are reflected off a surface. These are the surface's material properties. So there is a term to attenuate incident ambient light, one to attenuate incident specular light, etc. The eventual colour of a fragment is a function of the ambient light in the model reflected off the surface of the primitive, the ambient, specular, and diffuse components of individual (spot)lights reflecting off the surface of the primitive, and the emission of light from the surface of the primitive. The individual R, G, and B values for the colour of a fragment are calculated separately, and the alpha value is derived from the material's diffuse alpha value at the surface.

3.4 The OpenGL immediate and stored modes

OpenGL is able to function in two states regarding the way it handles the invocation of calls made to it — stored, and immediate mode.

3.4.1 Immediate mode

As a state machine, OpenGL interprets data according to the last state variables it was given (e.g. what colour subsequent vertices are associated with, whether vertices should be interpreted individually as points, or in triplets as triangles, etc.) In immediate mode, the OpenGL state machine draws to the screen, operations (and states) are not stored and therefore not recoverable.

3.4.2 Stored mode

The alternative to immediate mode is stored mode, which utilizes the concept of *display lists*. A display list may be thought of as a container in which may be placed OpenGL function calls (with some exceptions). It is possible to execute OpenGL function calls at the same time as placing them in a display list, but the real power of the display list is that it can be executed after creation (invoking all the OpenGL function calls that were placed inside it) without the overheads of any processing that was performed in obtaining those function calls.

In the client server model of OpenGL, display lists may become resident on the server side (in an analogous fashion to XWindows resources — see chapter 4) and so when operating in a client server environment, the increase in rendering performance of OpenGL in stored mode will be most noticeable. Indeed if the contents of a display list are currently sufficient to describe some scene except for a change in, say, the viewpoint, then only the new projection matrix need be transmitted from client to server in order for OpenGL to render the new view.

The disadvantages to using display lists are that they take up memory (which may be a scarce resource on the server where they will be stored) and are unmodifiable after creation. The latter restriction is present to minimize the memory management capabilities required of the server [15] where they will be stored. However, intelligent use of display lists and the execution of display lists from within other display lists give back the OpenGL programmer a degree of flexibility.

3.5 OpenGL drawing buffers

As vertices and states are defined to OpenGL, rendering may be performed in one or more of the four types of buffer that OpenGL makes available to the programmer. These buffers may be resident in hardware or software and each represents a rectangular array of pixels. Together, they comprise the framebuffer. Each pixel in a given buffer holds the same amount of information but pixels of different buffers may contain different amounts of information. The role of each buffer is briefly described below.

3.5.1 Colour buffer

The colour buffer pixel data can contain either a colour index (if OpenGL is operating in colour index mode [1]) or RGBA information (in RGBA mode operation).

There may be as many as four auxiliary colour buffers as well as front/back and left/right colour buffers. The auxiliary buffers are non-displayable but may be copied into one of the displayable front/back, left/right colour buffers. The intended use

[1]Colour index mode is a state in which a palette of colours is allocated by OpenGL before any rendering occurs. The colour of each rasterized pixel may only be selected from those colours available in the palette, each of which is referred to by a unique index.

of the left/right buffers is to produce stereoscopic views. Where front/back buffers exist, the front buffer(s) are displayed, enabling the programmer to render in the back buffer before swapping the two over. This technique is useful for producing smooth animation as the user sees the display updated in one go rather than it being added to gradually (as may be observed if OpenGL rendered directly to the front colour buffer).

3.5.2 Depth buffer

Pixel data in the depth buffer consists of a single floating point (or double precision, depending on the OpenGL implementation) number in the range ± 1.0. The contents of the OpenGL model are transformed by the projection matrix into the NDCS and then the z_{NDCS} value for each part of the model that is mapped into some region (x_{NDCS}, y_{NDCS}), $(x_{NDCS}+\frac{2.0}{pixels_x}, y_{NDCS}+\frac{2.0}{pixels_y})$ may be compared with the existing z_{NDCS} value for that pixel. The action taken upon the results of the comparison may be defined by the OpenGL programmer and can affect what gets drawn into the framebuffer. Most commonly, the depth buffer may be used to ensure that depth ordering is preserved in the model.

3.5.3 Stencil buffer

The stencil buffer may be used to limit what gets drawn into the framebuffer. Each pixel is represented in the stencil buffer (if enabled) by some number of bits (dependent on the OpenGL implementation but often the same number as allocated to an int). The pixel data values held in the stencil buffer can be used to restrict drawing to the framebuffer in a number of ways and, commonly, the stencil buffer may represent a mask outside of which (for example) no drawing will occur. The stencil buffer is also commonly used to achieve wireframe drawing with the appearance of the 'hidden' lines having been removed.

3.5.4 Accumulation buffer

The accumulation buffer may contain RGBA data (as does the colour buffer in RGBA mode) and is used to aggregate a series of images into a final image. The accumulation is not drawn to directly but may be used to transfer data to or from the colour buffer. It is commonly used to produce 'motion blur' or antialiasing effects. The operation of the accumulation buffer whilst OpenGL is in colour index mode is not defined.

3.6 Rendering Modes

The OpenGL API has been designed so as not to restrict its use merely to the production of images on a computer display. The state machine is able to be placed in one of three rendering modes, which dictate what happens to the NDCS vertices which OpenGL produces in its modeling phase. In the spirit of OpenGL, these modes do not offer the application developer functionality on a plate, but rather, they enable an application to evolve into an interactive, feedback driven process, able to operate on the model in true NDCS space, before vertices are mapped (and approximated) into a screen viewport. The three rendering modes are described briefly below, and possible uses for them in the Geant4 OpenGL graphics system drivers are discussed.

3.6.1 `GL_RENDER` mode

By making the function call `glRenderMode (GL_RENDER);`, OpenGL is placed in its default rendering mode. In this mode, all the NDCS coordinates are passed to OpenGL's rasterization phase to form viewport coordinate fragments. Before OpenGL is placed in `GL_RENDER` mode, a suitable connection to the current windowing system (Win32, XWindows, etc.) must have been made. `GL_RENDER` is the only rendering mode that affects the contents of the framebuffer.

3.6.2 `GL_SELECT` mode

The `GL_SELECT` rendering mode enables an application to establish which components of a model fall within a particular viewing volume. When drawing in `GL_SELECT` mode, each primitive may be associated with a name. The name associated with a primitive is that which was on the top of the *name stack* when the primitive was defined. The name stack can be manipulated in ways similar to the other stacks that OpenGL uses (modelview matrix stack, etc.,) by pushing, popping, initializing and loading the stack, and is composed of `unsigned int`s. It is just a vehicle for associating some (shared or unique) identifier with a primitive geometrical object. If a stack remains unchanged between drawing two or more primitives while in `GL_SELECT` mode, then those primitives share the same name.

Drawing in `GL_SELECT` mode proceeds in much the same way as it does in any other rendering mode. Furthermore, the commands which associate primitive objects with names and manipulate the name stack are simply ignored by OpenGL if it is not in the `GL_SELECT` state, so the same piece of application code can be employed to perform drawing in any of the rendering modes. The (orthographic or perspective) viewing volume that is defined whilst in `GL_SELECT` mode is that with which drawn primitives must overlap if their drawing in `GL_SELECT` mode is to produce a *hit record*. A hit record consists of four components, and is generated only when the name stack is manipulated, or the rendering mode is changed. The four components of each hit record are the number of names in the name stack at the time of the hit, the minimum scaled[2] NDCS z coordinate of all vertices of all primitives overlapping

[2]Minimum and maximum NDCS z coordinates (in the range [-1.0, 1.0]) get multiplied by $2^{32}-1$ before they are written in a hit record.

the viewing volume at the time of the hit, the maximum scaled NDCS z coordinate of all vertices of all primitives overlapping the viewing volume at the time of the hit, and the contents of the name stack at the time of the hit. Hit records get written to the *select buffer*. The select buffer is an array of `unsigned int` objects which must be allocated and declared to OpenGL before entering `GL_SELECT` mode. The number of hit records written to the select buffer by OpenGL in `GL_SELECT` mode is returned from `glRenderMode(...)` when the rendering mode is changed, and may may of use when reading the select buffer for hits records.

Picking in `GL_SELECT` mode

Picking is a useful extension of selection, whereby the selection viewing volume is interactively chosen by the application user. If a user is trying to focus attention on a particular component of the model described to OpenGL, then they may do so via `GL_SELECT` mode. By using the `gluPickMatrix(...)` function call, an application can receive information from the user (via a mouse point-and-click operation, for example) on the region that the user is interested in (in screen coordinates), and define the viewing volume for `GL_SELECT` mode drawing to be around this point (in NDCS coordinates). The `gluPickMatrix(...)` function takes arguments to describe the x and y screen coordinates (returned by the mouse point-an-click, for example), the width and height (in screen dimensions, i.e. pixels), and the current OpenGL viewport. It then postmultiplies a special projection matrix onto the current projection matrix to create the desired viewing volume for OpenGL to use in `GL_SELECT` mode. In this way, the sensitivity of picking may be defined by the application (through the width and height arguments) and the number of hit records generated in `GL_SELECT` mode can be minimized, hastening the determination of what element of the model the user was actually trying to pick.

Uses of `GL_SELECT` rendering in Geant4

The potential uses of the `GL_SELECT` rendering mode within the Geant4 OpenGL drivers are mostly in the delivery of user requirements UR10-2 and UR10-5 (see table 5.1). At the time of writing, the general Geant4 visualization framework for picking and user selection is in the process of being designed, and no attempt will be made to incorporate picking and user selection into the OpenGL drivers for Geant4 until such a framework exists for all graphics systems. Eventually, it is anticipated that a Geant4 user will be able to interrogate the Geant4 kernel about individual elements of the model visualized in Geant4, via the picking mechanism. The user will then be able to enquire after the amount of energy deposited in a sensitive detector, the mother particle of a track, the track associated with a particle and many other quantities via picking.

3.6.3 `GL_FEEDBACK` mode

The `GL_FEEDBACK` rendering mode in OpenGL allows an application to inspect the primitives that are produced in representation of the model described to OpenGL. This occurs instead of rasterization, and so the framebuffer is unaltered while OpenGL is in this state. Before the application enters `GL_FEEDBACK` mode (via `glRenderMode(GL_FEEDBACK);`), a *feedback buffer* must be defined in the application and declared to OpenGL via the `glFeedbackBuffer(...);` function. This function includes an argument that specifies how much NDCS vertex information to write to the feedback buffer. The options are given in table 3.1.

The primitives written into the feedback buffer are separated by a code indicating the primitive type. A full list of these is given in table 3.2. It should also be noted that the primitives written into the feedback buffer are those that would otherwise be

Type argument	Coordinates	Colour	Texture	Total values
`GL_2D`	x, y	none	none	2
`GL_3D`	x, y, z	none	none	3
`GL_3D_COLOR`	x, y, z	k	none	3+k
`GL_3D_COLOR_TEXTURE`	x, y, z	k	4	7+k
`GL_4D_COLOR_TEXTURE`	x, y, z, w	k	4	8+k

Table 3.1: Possible feedback modes in OpenGL. If OpenGL is in RGBA mode then k=4, and if OpenGL is in Colour Index mode, then k=1.

rasterized by OpenGL. By the rasterization phase, depending on the implementation of OpenGL, primitives with more than three vertices may have been decomposed into triangles (as, for example, hardware may be able to rasterize these optimally). To avoid confusion over the nature of primitives in the feedback buffer, the function `glPassThrough(GLfloat` *token*`);` may be used between primitives being defined to OpenGL by the application. This function effectively inserts a primitive of type `GL_PASS_THROUGH_TOKEN` followed by *token* to act as a marker in the feedback buffer, which can be interpreted by the application. This can help the application parse the feedback buffer, and removes any uncertainty over whether a list of triangles in the feedback buffer represents a list of triangles defined to OpenGL in the application, or the decomposition into a list of triangles of some other polygon.

Uses of `GL_FEEDBACK` rendering in Geant4

As `GL_FEEDBACK` mode rendering eliminates the rasterization phase of OpenGL, one of the most obvious applications in Geant4 is towards the production of high quality, device resolution independent hardcopy output. Older graphics systems

Primitive type	Enumeration	Associated data
Point	`GL_POINT_TOKEN`	vertex
Line	`GL_LINE_TOKEN` or `GL_LINE_RESET_TOKEN`	vertex, vertex
Polygon	`GL_POLYGON_TOKEN`	n vertex, vertex, ...
Bitmap	`GL_BITMAP_TOKEN`	vertex
Pixel rectangle	`GL_DRAW_PIXEL_TOKEN` or `GL_COPY_PIXEL_TOKEN`	vertex
Pass-through	`GL_PASS_THROUGH_TOKEN`	GLfloat

Table 3.2: Types of primitive found in the feedback buffer.

such as PHIGS[3] offer PostScript model description as a 'built in' feature, but the `GL_FEEDBACK` rendering mode in OpenGL exceeds this capability by not restricting the possibilities for high quality hardcopy to PostScript. If required, a processing engine that is able to produce line plotter files, or even CAD[4] STEP[5] files could be connected, although the latter possibility may be approaching computer modeling from the wrong direction.

Currently, the Geant4 OpenGL graphics system drivers offer the user the option to write colour or greyscale vectored PostScript files (via `GL_FEEDBACK` mode rendering (see section 5.8.5).

[3]A very primitive version of the Programmers Hierarchical Interactive Graphics System (PHIGS) drivers for Geant4 were written in the very early development stages of Geant4 visualization, to aid the evolution of the general graphics system framework.

[4]Computer Aided Design.

[5]STandard for the Exchange of Product model data.

Chapter 4

Xlib and Motif

The X Windowing System (XWindows) is a platform-independent software specification for displaying and manipulating graphics and text in a network environment, developed jointly by the Massachusetts Institute of Technology Laboratory for Computing Science and MIT's Project Athena. The first widely used version of XWindows was version 10.4, released in 1986, but more often used today, are versions 11.5 (X11R5) and 11.6 (X11R6) which were released in 1994. The system was originally created to extend the set of data objects that could be efficiently communicated between computers via a network beyond mere ASCII characters, and it is now commonly employed in Unix-based environments. It has been used in Geant4 to provide a mature software layer between the graphics hardware and OpenGL libraries (see figure 4.1).

4.1 The XWindows client-server model

One of the initial goals of XWindows was to enable many different types of machines to cooperate via a network. To do this there must be processes running on both the machine that controls the display and input devices (*server*) and on the machine issuing the requests (*client*) that understand the XWindows protocol. Some elements of the XWindows protocol and their interaction with the G4OpenGL[X/Xm] drivers are described briefly below.

4.2 Features of the XWindows protocol

4.2.1 Buffering

XWindows attempts to decouple the client and server processes from the availability of access to the network. A request sent by the client is only transmitted immediately if it requires an immediate response from the server or if the client asks the request buffer to be flushed manually. Also, if the client issues a request to wait until a certain XWindows event happens, then, given the uncertain time before the event may occur, the request buffer is flushed. This makes XWindows tolerant of transient network availability. However, caution must be exercised in flushing the buffer so as neither to remove the benefits of buffering, as might happen if the buffer is flushed by the user after every piece of XWindows traffic, nor to allow so much to accumulate in the buffer that the information displayed to the user is misleading, as may happen if OpenGL requests lie dormant in the XWindows buffer after the drawing appears to have finished.

4.2.2 Resources

The communication of windowing information via XWindows requests represents a reduction in the volume of information that has to be transmitted between server and client over the pixel-by-pixel descriptive method. Another way to help reduce this network traffic is by placing some compound items, such as fonts, pixmaps or colormaps, in the server memory. These items can then be allocated some form of integer ID so that the client can refer to the items again in an efficient manner rather than having to retransmit all of the information they contain. Furthermore, any client can use any of the resources stored on a server if it knows the resource's identifier There is no concept of the ownership of a resource between clients.

An example of a resource reused by `G4OpenGLX` code is the colormap. A colormap is a C `struct` that XWindows uses to describe the range of colours that are available for drawing in. An XWindows server has the concept of a default RGB colormap and a best RGB colormap, whose existences may be tested for. Often, the window manager (see section 4.2.3) will have already created one or other of these. If a suitable colormap already exists on the server then `G4OpenGLXView` simply uses it as there is no need to allocate a new one. If the server has yet to allocate a default or best RGB colormap, or if the ones allocated are insufficient for OpenGL to use, then `G4OpenGLX` code will attempt to allocate a new colormap.

4.2.3 The window manager

XWindows has the concept of a window manager. This is a process that runs on the server and may receive and process requests made to it by the client. There are several advantages to forcing all XWindows instructions through a window manager. If multiple client processes attempt to display output on the same XWindows server then the window manager is responsible for maintaining a single copy of each resource (font, colormap, pixmap, etc.) needed by the clients. Also, the window manager decides where and how to place the windows requested by the clients, resolving any conflicts that may arise.

4.3 XWindows Toolkit Intrinsics

XWindows Toolkit Intrinsics (Xt) is built on top of XWindows (Xlib) and introduces the concept of an event driven process. An event is defined in XWindows as an asynchronous notification that something of interest to the client process has happened at the server [9].

In a conventional XWindows environment, a process must prompt a user for input in order to determine the direction of program flow. In an event driven environment, a process can constantly expect user input and respond to it accordingly, producing a more natural user-process interaction. The cost of this extra interactivity is an infinite event getting and processing loop.

One potential problem with the infinite event getting loop is the dedication of all processor time to running this loop. This lends itself to 'locking out' all other processes for the duration of the polling loop. A partial solution to this has been used within Geant4 visualization by manufacturing a single XWindows event getting loop and registering each XWindows event producing process with the `G4Xt` (see section 5.8.6) singleton object. This object has responsibility for gathering XWindows events and dispatching them to the relevant process. By doing so, Geant4 is able to have any number of XWindows event driven graphics systems (or interfaces) coexisting without locking each other out. The solution does not however provide an answer for how to avoid locking out all other Geant4 processes (tracking, hits scoring, etc.) when polling for XWindows events is taking place.

Provision is made in Xt for the creation and maintainance of widgets. A widget is an object that the window manager is able to identify and associate with events (in the XWindows sense). Typically, an event driven process creates, maps, and realizes (i.e. puts on screen) many widgets to make an interactive window. Each button, scroll bar, menu, dialogue box, etc., is a different widget and, upon creation at the XWindows server, is assigned a 32 bit identifier Also associated with each widget is

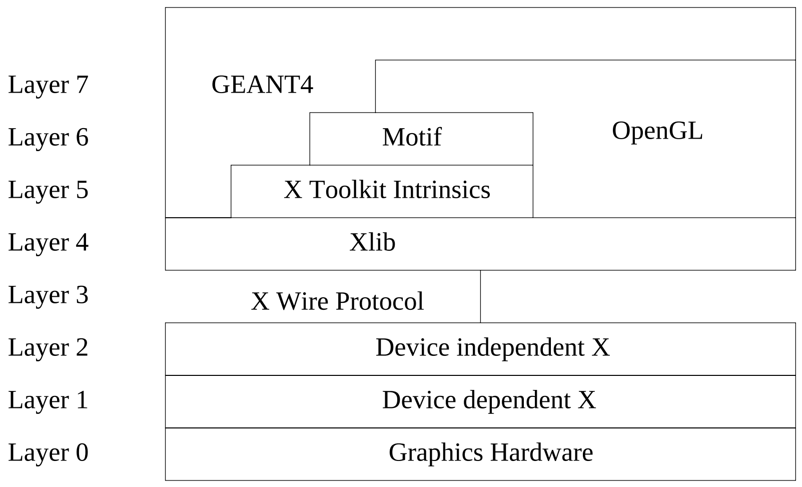

Figure 4.1: Relations of Motif, Xt and XWindows.

an event mask, which advises the window manager on which event types (the basic XWindows server reports 34 different types of event) are of interest to the client for that widget. When a relevent event occurs in a widget, the window manager sends an event notification to the client that created it. In the interests of performance, both the server-to-client event stream and the client-to-server request stream are usually asynchronous, i.e., the XWindows server does not wait for a reply to one event before sending the next event.

4.4 The G4OpenGLXm widget wrapper classes

There exist many widget sets such as Motif (Xm) and Athena (Xaw) that attempt to form more useful objects such as scroll bars and menu bars out of the Xt library. The Geant4 toolkit contains OpenGL drivers interfaced to Xm (`G4OpenGLXmView`) to provide an interactive OpenGL environment. In an attempt to help `G4OpenGLXmView`'s functionality to evolve with Geant4's capabilities, a number of classes have been written to wrap the Xm widgets and their interactions so that future developers need no Xm expertise to modify, for example, the Geant4 control panels made available to the `G4OpenGLXmView` user. There are many commercial graphical user interface (GUI) building packages available (VUIT, Xdesigner, VXP, etc.), which is why the decision was taken to provide the `G4OpenGLXmView` classes. Given the timescale over which `G4OpenGLXmView` must perform (LHC and beyond) and the uncertainty whether a future Geant4 user will have access to exactly the right GUI builder if any, it seemed sensible to supply a robust way of adding/subtracting buttons, sliders, etc. as part of the `G4OpenGLXmView` distribution. The nine classes that currently comprise the `G4OpenGLXmView` Xm wrapper classes are described below.

4.4.1 Overview

The procedure for amalgamating buttons, bars, etc., into meaningful and useful control panels, dialog boxes, etc., in Xm is to construct a hierarchy of widgets, such that each widget is parented by some other. The highest widget in this hierarchy is derived from the window manager. In this way, it is possible to propagate data essential to the life of every widget, such as colormaps and other data relating to the XWindows server they are being displayed on, to all widgets consistently. In the author's experience, this does not happen automatically, as the literature suggests it should, resulting in frequent and fatal XWindows errors. Fortunately, all these automatically inherited resources can be set by the user.

One job performed by the Xm wrapper classes is to ensure that all critical widget data are synchronized with each other. This is achieved by having all `G4OpenGLXmView` wrapper classes inherit from `G4OpenGLXmVWidgetObject`, a class that acts as a repository for critical widget data. All these data are derived from the original connection made between `G4OpenGLXView` and the XWindows server. This process ensures the avoidance of XWindows errors relating to badly set or mismatched widget resources.

4.4.2 Shells

In Xm, a shell is the point of contact between the window manager and a (collection of) widget(s). Only one type of shell has been implemented in the `G4OpenGLXmView` wrapper classes. This corresponds to the most flexible of Xm shells, the TopLevel shell.

(i) G4OpenGLXmTopLevelShell

A top level shell may become decorated by the window manager with a border, name, minimize button, etc., and must conform to the window manager's rules of layout on the screen. The `G4OpenGLXmTopLevelShell` inherits from `G4OpenGLXmVWidgetShell` which, via pure virtual functions, ensures that any derived class provides methods to add child widgets, return a pointer to itself, and realize itself on the screen.

4.4.3 Containers

There are two `G4OpenGLXmView` wrapper class containers, which are described below. Their job is to manage any components placed within them. In the interests of simplicity, only classes derived from `G4OpenGLXmVWidgetComponent` may be placed within a container. The parent class, `G4OpenGLXmVWidgetContainer`, ensures, via pure virtual functions, that any derived class provides methods to add component widgets to itself and add itself to a shell.

(ii) G4OpenGLXmBox

The `G4OpenGLXmBox` class implements the RowColumn Xm widget in `G4OpenGLXmView`. This container arranges its children to make best use of its available area and is able to act as a container for radio buttons (see below). If it is to act as such then it is guaranteed that only one of all the radio buttons contained will be active at any one time.

(iii) G4OpenGLXmFramedBox

The `G4OpenGLXmFramedBox` is simply an Xm RowColumn widget (such as is created by `G4OpenGLXmBox`) placed inside an Xm FrameWidget. The effect is to place a visible border around the container. It is useful for visually compartmentalising components to ease understanding of a control panel's functionality.

4.4.4 Components

A lot of the effort that has been made towards the simplification offered by the XmWrapper classes has been invested in those classes that are derived from the base class `G4OpenGLXmVWidgetComponent`. They attempt to provide functionality to the user in an uncomplicated yet safe way.

(iv) G4OpenGLXmFourArrowButtons

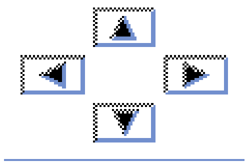

Figure 4.2: The `G4OpenGLXmFourArrowButtons` class widget

This class creates an Xm FormWidget object, in which are placed four ArrowButtonGadgets pointing up, down, left, and right. The responses offered by selection of each button are defined in a callbacks list that must be specified upon creation of the object. Multiple callback functions may be registered with each button.

(v) G4OpenGLXmPushButton

The `G4OpenGLXmPushButton` class creates a simple widget button, with some text printed on it, which may have multiple callback functions invoked by its selection.

(vi) G4OpenGLXmRadioButton

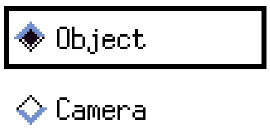

Figure 4.3: The `G4OpenGLXmRadioButton` class widget

The `G4OpenGLXmRadioButton` class creates a toggle button. Callback functions may be registered to be invoked by selection of the button and, if selected, it appears 'depressed' until deselected by the user. If a number of `G4OpenGLXmRadioButtons` are placed inside a `G4OpenGLXmBox` or `G4OpenGLXmFramedBox` then the container is able to manage them in such a way as either to allow many of the radio buttons to be selected at once, or exactly one. The default action is to allow many radio buttons to be selected concurrently.

(vii) G4OpenGLXmSeparator

The `G4OpenGLXmSeparator` class creates a vertical bar across a container. It is most useful for the purpose of visually dividing a control panel into functional parts.

(viii) G4OpenGLXmSliderBar

`G4OpenGLXmSliderBar` creates a horizontal or vertical scale, along which can be drawn a slider. The scale has a minimum, maximum, and initial value, which may be displayed immediately above or to the side of the slider. The Xm slider bar is

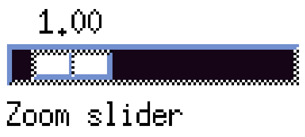

Figure 4.4: The `G4OpenGLXmSliderBar` class widget

only able to represent integer values so an order of magnitude F may be supplied to the constructor, which has the net effect of multiplying the minimum and maximum values supplied by 10^F and dividing the displayed value (if requested) by the same factor, giving the same appearance as a slider that is able to accommodate non integer data.

(ix) G4OpenGLXmTextField

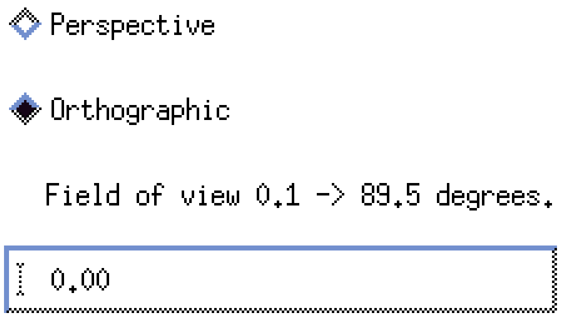

Figure 4.5: An example of the `G4OpenGLXmTextField` class widget with two objects of type `G4OpenGLXmRadioButton` above it governing the interpretation of input

The `G4OpenGLXmTextField` class enables a Geant4 user to enter data directly to `G4OpenGLXmView`. The data is internally represented as a text string (whether it is really a text string or representative of numeric data). The variable into which user entered data is written is specified in an overloaded constructor, which is responsible for determining whether it is relevant to interpret input as text or numeric data.

4.4.5 Example use of the G4OpenGLXm wrapper classes

The intended purposes of the `G4OpenGLXmView` wrapper classes are threefold:
(i) to propagate widget resources properly and reliably;
(ii) to simplify the inclusion of new features of Geant4 into `G4OpenGLXmView` code;
(iii) to isolate future developers from the need for any detailed Xm knowledge.

To illustrate some of the power of the classes, a brief code fragment is described below. The code fragment is called from within a derivative of the `G4OpenGLXmView` class and creates a control panel to offer the user the opportunity to create an encapsulated postscript (EPS) file of the current `G4OpenGLXmView`.

```
G4OpenGLXmTopLevelShell* shell =
new G4OpenGLXmTopLevelShell (this, "Printing control");

G4OpenGLXmFramedBox* button_box =
new G4OpenGLXmFramedBox ("Create an EPS file", False);
shell->AddChild (button_box);

G4OpenGLXmFramedBox* colour_box =
new G4OpenGLXmFramedBox ("Colour choice", True);
shell->AddChild (colour_box);

G4OpenGLXmFramedBox* style_box =
new G4OpenGLXmFramedBox ("File type", True);
shell->AddChild (style_box);
```

The first action performed is to create `shell`, a `G4OpenGLXmTopLevelShell*` for the window manager to communicate with. The first argument to `shell` is of type `G4OpenGLXmView*`, and contains all the information that is required by the `G4OpenGLXmTopLevelShell` object to derive the necessary widget resources. Next, `button_box`, `style_box` and `colour_box`, all of which are `G4OpenGLXmFramedBox*`s are created and given to `shell` to manage. The second argument to the constructor of `G4OpenGLXmFramedBox` specifies whether the container is going to be required to act as a radio button box or not.

The following three code segments place useful `G4OpenGLXmVWidgetObject`s in these three `G4OpenGLXmVWidgetContainer`s, in order to create a useful 'popup' control panel.

```
G4OpenGLXmTextField* text =
new G4OpenGLXmTextField ("Name of file", (pView->eps_string));
button_box->AddChild (text);

G4OpenGLXmSeparator* line = new G4OpenGLXmSeparator ();
button_box->AddChild (line);

XtCallbackRec* cb_list = new XtCallbackRec[2];
cb_list[0].callback = G4OpenGLXmView::print_callback;
cb_list[0].closure = pView;
cb_list[1].callback = NULL;

G4OpenGLXmPushButton* button =
new G4OpenGLXmPushButton ("Create EPS file", cb_list);
button_box->AddChild (button);
```

The first argument to the `G4OpenGLXmTextField` constructor defines what text is to be printed above the text entry area. `text` allows the user to choose the name of the file that will be written. The second argument to `text` is of type `char*` (although it could just as well be `double`) and relates to the variable in which to store whatever gets typed into the text widget. The second argument is also what is displayed initially in the text entry area.

`line` is added to `button_box` simply to create a visual separation between the `G4OpenGLXmTextField` widget and the `G4OpenGLXmPushButton` below it. `cb_list` is an array that contains all the callback functions that are to be invoked upon selection of a particular widget. In this case, only one callback, the static class member `G4OpenGLXmView::print_callback`, is registered and the list is `NULL` terminated. The second array element, `pView`, is an argument passed to the callback function. `cb_list` is used as the second argument to the `G4OpenGLXmPushButton` constructor, to define what actions are carried out upon selection of `button` by the user. The first argument to the `G4OpenGLXmPushButton` constructor defines the text to display on the button.

```
cb_list[0].callback = G4OpenGLXmView::set_print_colour_callback;
cb_list[0].closure = pView;
cb_list[1].callback = NULL;

G4OpenGLXmRadioButton* colour_radio1 =
new G4OpenGLXmRadioButton ("Greyscale", cb_list, False, 0);
colour_box->AddChild (colour_radio1);

G4OpenGLXmRadioButton* colour_radio2 =
new G4OpenGLXmRadioButton ("Colour", cb_list, True, 1);
colour_box->AddChild (colour_radio2);
```

Next, two radio buttons, `colour_radio1` and `colour_radio2`, are created and managed by `colour_box`. The four arguments to `G4OpenGLXmRadioButton` relate to: the text string to display with the radio button; the callback function(s) to associate with selection of the button; whether the button is initially selected or not; and an integer identifier to associate with selection of the button.

```
cb_list[0].callback = G4OpenGLXmView::set_print_style_callback;
cb_list[0].closure = pView;
cb_list[1].callback = NULL;

G4OpenGLXmRadioButton* style_radio1 =
new G4OpenGLXmRadioButton ("Screen dump (pixmap)", cb_list,
False, 0);
style_box->AddChild (style_radio1);

G4OpenGLXmRadioButton* style_radio2 =
new G4OpenGLXmRadioButton ("PostScript", cb_list, True, 1);
style_box->AddChild (style_radio2);

shell->Realize();
```

In adding `G4OpenGLXmVWidgetObjects` to `style_box`, the same methods as were used to fill `colour_box` are employed. After `button_box`, `colour_box`, and `style_box` have been filled with whatever `G4OpenGLXmVWidgetObjects` are required, the window manager can be requested to manage and display all the widgets by issuing the `Realize()` method of `shell`.

The control panel created enables the Geant4 OpenGLXm user to create PostScript files of the image currently rendered in OpenGL, and its displayed appearance is shown in figure 4.6. The details of the process of PostScript file production in OpenGL are discussed in section 5.8.5.

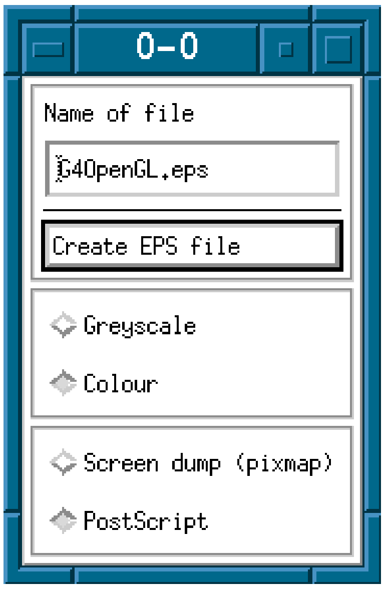

Figure 4.6: The printing control panel in `G4OpenGLXmView`

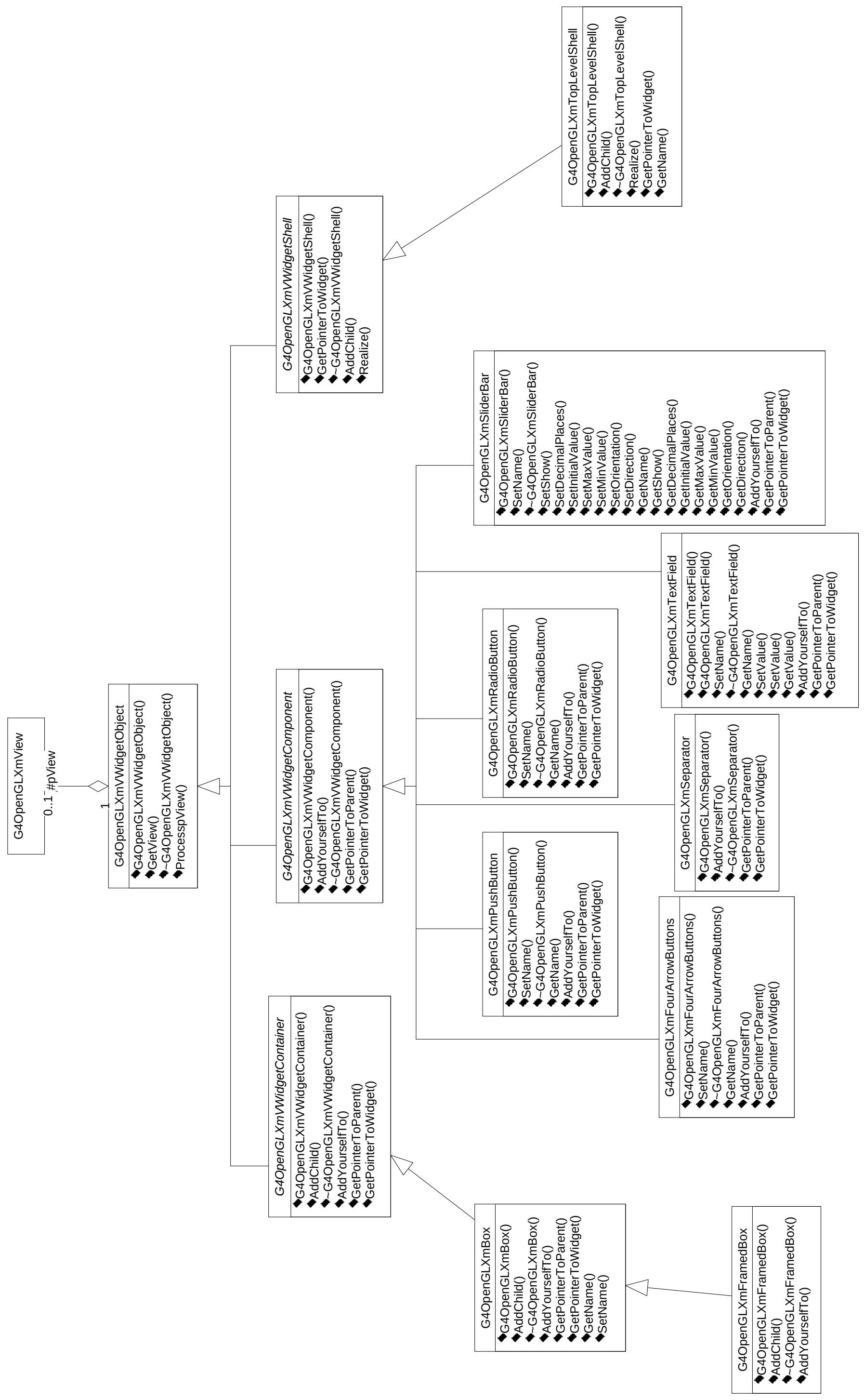

Figure 4.7: A class diagram of the Xm wrapper classes

Chapter 5

Geant4 Visualization

The goals of the visualization component of the Geant4 toolkit are stated in the Geant4 User Requirements Document [1], and are summarized in table 5.1. The philosophy is to provide an item of the Geant4 toolkit which may be easily tailored and incorporated into a user's application, in order to satisfy their specific visualization needs.

Item	Requirement for visualization	Implemented
UR 10-1	Entire detector geometry setup or subsections thereof	✓
UR 10-2	Parameters of individual geometrical detector entities	✗
UR 10-3	Particle trajectories at each step of tracking	✓
UR 10-4	Detector response in sensitive detector elements	✓
UR 10-5	Navigation of genealogy of event information	✗
UR 10-6	Genealogy relationships of detector geometry structure	✗
UR 19-7	Well defined interface to visualization. User able to build a visualizer of their choice into the framework	✓

Table 5.1: User requirements of visualization in Geant4.

The visualization needs of a user can vary between none if they require Geant4 to be a non-interactive batch process for the production of simulation data, and heavy if they would like their application to enable them to navigate a virtual model of the detector and view particle trajectories from any angle. The graphics systems currently implemented in the Geant4 visualization framework are able to produce detector visualizations quickly on inexpensive computers, create VRML files which can be viewed in a virtual reality browser, allow the user to interact with the model of the detector, and produce very high quality hardcopy output. It is up to the user to decide how much of the available functionality they wish to incorporate into their application.

5.1 User interface commands for visualization

The following sections describe in some detail the inner workings of Geant4 visualization. The Geant4 user is not expected to be conversant in such matters, but is instead offered a (growing) set of commands via a User Interface (UI) session. The components of the Geant4 toolkit are able to register commands with the UI manager which hopefully will satisfy most end users. The Geant4 user is confronted with a UI session (which may be a simple command line interface, or a more friendly and dynamically built interface panel such as GAG[1]) from which they may issue these commands. Each command is an object in a hierarchical structure. Each object in this structure is of a subclass of `G4UICommand`, and is instantiated in the constructor of a subclass of `G4UImessenger`. Methods to carry out the command are defined in the particular subclass of `G4UImessenger`.

The expression syntax of commands in the UI session is similar in structure to a unix command line. To draw the current scene in the current viewer, the user may

[1]GAG is the Geant4 Adaptive Graphical Java-based or Tcl/Tk-based user interface developed in Naruto, Japan.

issue a command such as `/vis/refresh/view` (or click the equivalent button in the GAG interface).

5.2 The Geant4 interface to visualization

Visualization is one component of the Geant4 toolkit. In order to satisfy the requirement that it does not form an integral part of the Geant4 kernel[2] there must exist an abstract interface to visualization within the kernel, which must remain entirely independent of any visualization drivers. This interface is formed by the classes `G4VVisManager` and `G4VGraphicsScene`. Both of these classes declare pure `virtual` functions (see section 2.2.1) which are able to declare those functions which must be supplied by any derived classes (in the visualization side of the interface) and resolve any references made to the visualization toolkit in user code [3]. User code may refer to elements of visualization say, to draw a red dot at every point where there is a hit in the detector. To avoid the need to rewrite and recompile user code between building visualization-inclusive and visualization-exclusive applications, it is sufficient to protect the user code with the condition that a suitable derived class of `G4VVisManager` is instantiated and therefore able to handle requests for visualization. An example of such protection is given in figure 5.1.

Initially, requests for visualization are directed towards the abstract visualization interface which is embedded in the Geant4 kernel. If the visualization component forms a part of the application, then calls made to functions in the abstract interface are realized — via the `virtual` function mechanism (see section 2.2.3) — as invocations of derived class member functions. Once the initial visualization request transactions have been carried out, objects of Geant4 visualization classes are able

[2]The kernel is that portion of Geant4 which is present in every Geant4 application, and is essential to communication between toolkit components.

[3]User code is that code which is supplied by the Geant4 user, and acts as a client of Geant4.

```
G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
if (pVVisManager) {
...Geant4 visualization code...
}
```

Figure 5.1: The protection of Geant4 user visualization code.

to query Geant4 to gather more information as they need it. Note that initially, Geant4 is the client, and the abstract visualization interface is the server; after the initial exchanges, the visualization component becomes the client, and the Geant4 kernel the server. This enables visualization objects to extract data from the Geant4 kernel in the form and manner of their choosing, which may be tailored to a particular graphics system's needs or abilities. Note also that the abstract interface to visualization avoids a circular dependence.

5.3 The interface to the graphics systems

The visualization component of the Geant4 toolkit is required not to be bound to any particular graphics system (UR 19-7). To achieve this, the three classes `G4VGraphicsSystem`, `G4VScene`, and `G4VView` are provided from which may be derived implementations for graphics systems which are referred to as graphics system drivers. It is inside these drivers (such as `G4OpenGLStoredX`, `G4OpenGLScene`, and `G4OpenGLView`) that the interpretation of Geant4 visualization requests for graphic-system-dependent operations takes place. A schematic representation of the communication between the kernel, visualization component, and graphics system is shown in figure 5.2.

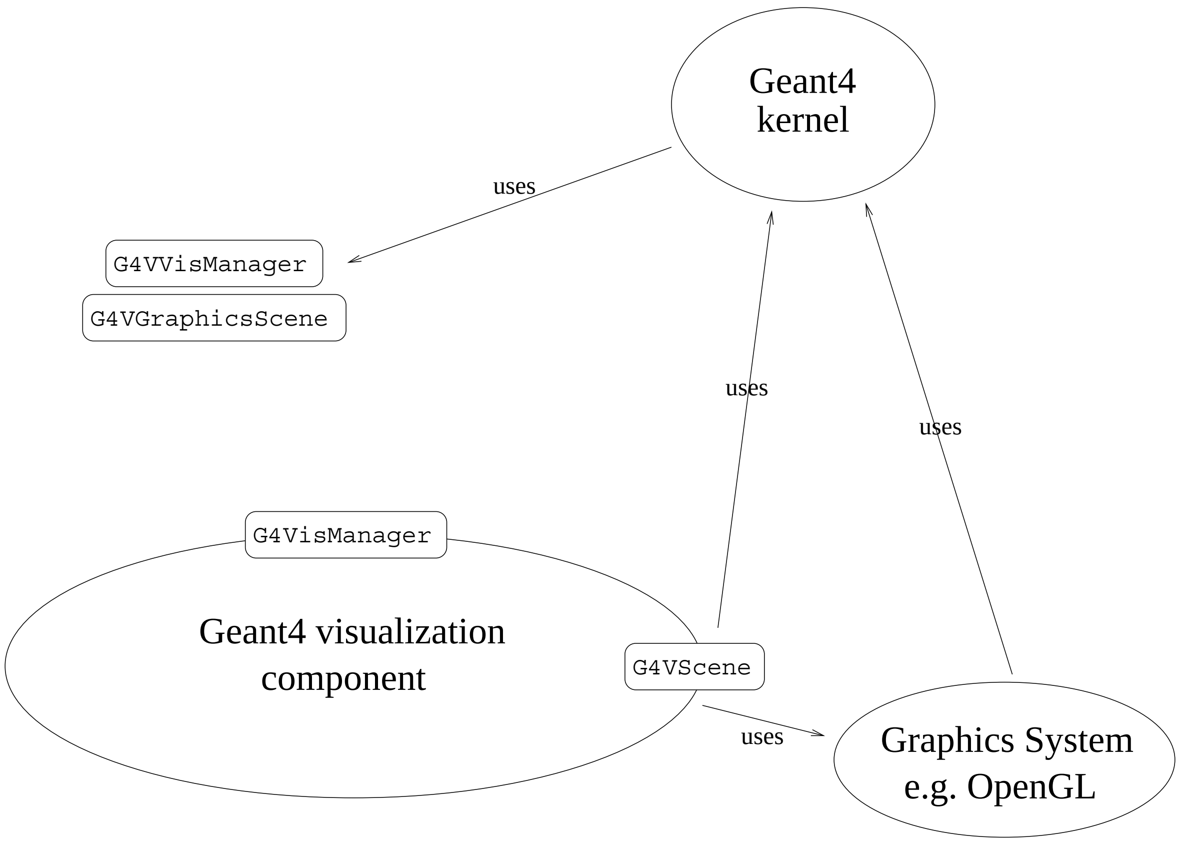

Figure 5.2: A schematic of communication for Geant4 visualization. The pure abstract interfaces break the circular dependency.

5.4 The implementation of a graphics system

To accomodate the full range of graphics functionality that is to be offered to the Geant4 user, a highly general interface to graphics systems has been sought. The essential features of any graphics system that uses a graphical database (the graphical database in this case being supplied by the Geant4 kernel) include an ability to create some model with which the visualization is concerned, and an ability to render that model, such that it becomes visualized. The abstract base class `G4VScene` is concerned with declaring the methods which will be utilised by derived classes in creating a graphics system dependent model of items in the Geant4 graphical database. The abstract base class `G4VView` declares the methods upon whose invocation in derived classes, the graphics system will render the model.

5.4.1 The `G4VGraphicsSystem` base class

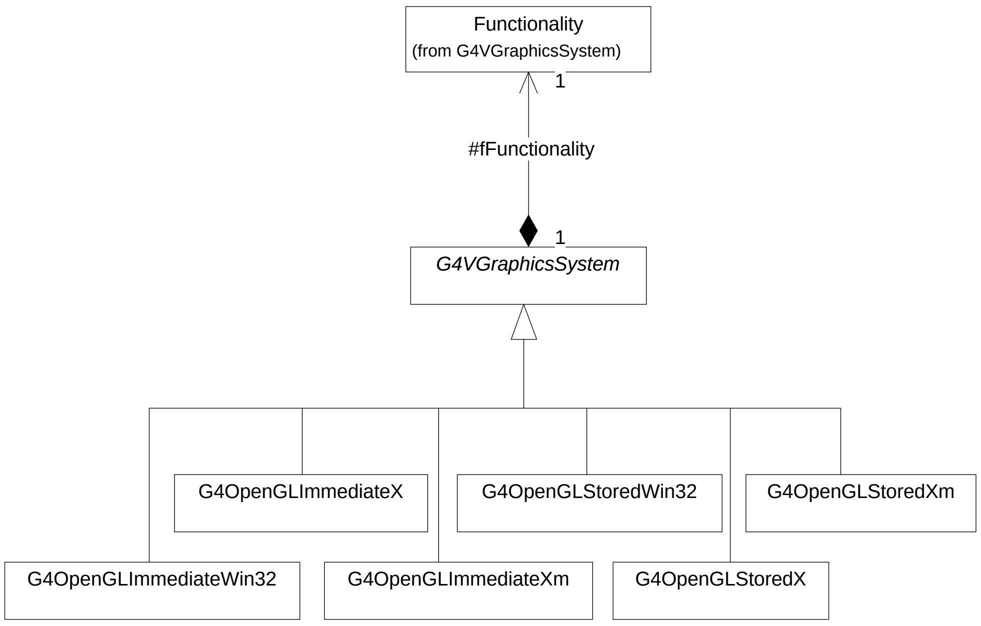

Figure 5.3: The OpenGL implementation of `G4VGraphicsSystem`

`G4VGraphicsSystem` declares pure `virtual` methods which when defined in a derived graphics system implementation class, are able to instantiate a scene and a view of that graphics system. Each implemented graphics system class derived from `G4VGraphicsSystem` which is to become a part of a given application must be registered with the user's `G4VisManager` object in the user's main program. The Geant4 distribution facilitates the inclusion of graphics driver code via the use of preprocessor `#ifdef` flags within source code. Examples of such are provided with the distribution. Such flags may be set before the application is built to dictate which graphics systems become registered (and hence, form a part of the application).

Upon registration, each `G4VGraphicsSystem` derived object is appended to a list of available graphics systems, and offered to the user.

```
virtual ~G4VGraphicsSystem ();

virtual G4VScene* CreateScene () = 0;

virtual G4VView* CreateView() = 0;
```

Figure 5.4: `virtual` function declarations of `G4VGraphicsSystem`

5.4.2 The `G4VScene` base class

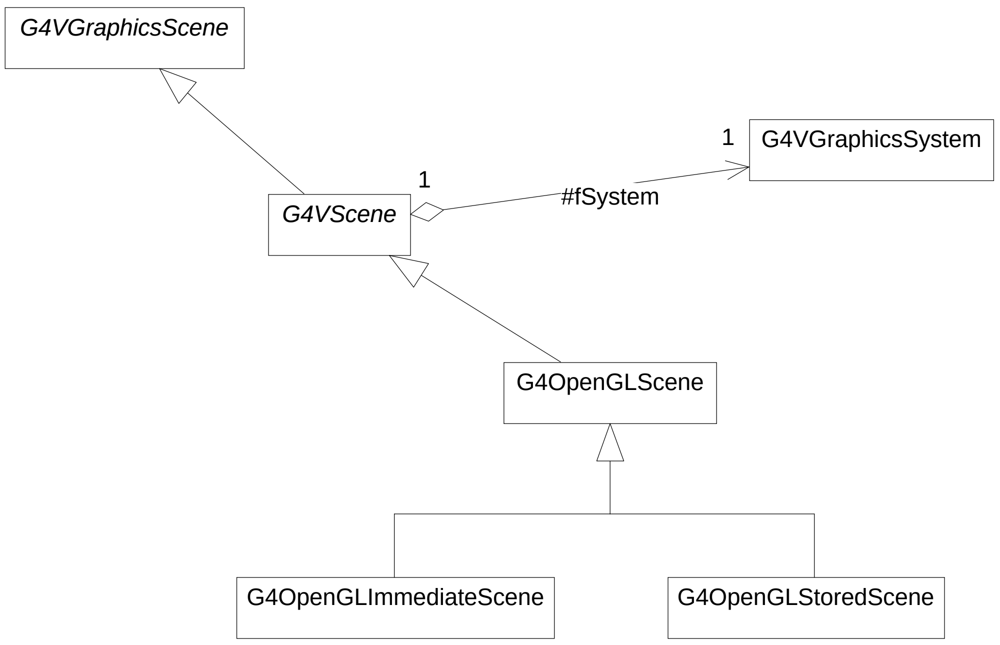

Figure 5.5: The OpenGL implementation of `G4VScene`

The `G4VScene` class inherits from `G4VGraphicsScene` (a member of the kernel) and declares those objects out of which it is reasonable for the Geant4 graphical database to compose a visualizable model. One salient feature of a `G4VScene` is its `G4SceneData` data member. This maintains a list of all the `G4VModel`s which are displayed by the scene to which the `G4SceneData` object belongs.

```
virtual void AddThis (const G4Box&);
...
virtual void AddThis (const G4VSolid&);
virtual void PreAddThis (const G4Transform3D&,
const G4VisAttributes&);
virtual void PostAddThis ();
virtual void BeginModeling ();
virtual void EndModeling ();
virtual void BeginPrimitives (const G4Transform3D&);
virtual void EndPrimitives ();
virtual void AddPrimitive (const G4Polyline&) = 0;
...
virtual void AddPrimitive (const G4NURBS&) = 0;
```

Figure 5.6: `virtual` function declarations of `G4VScene`

In Geant4, geometries are described in a hierarchical fashion, and a `G4VModel` has the concept of a 'top volume' to describe that volume which has no parent, most usually relating to the Geant4 'world'. The model is able to respond to a request by the scene to describe itself, by describing the top volume, before descending the geometry hierarchy, and describing each daughter volume encountered until some predefined genealogical depth is reached, or no more daughters are defined. For each volume encountered, an attempt is made to append the graphics system scene with the volume itself 'as is', i.e., as a `G4Box`, `G4Tubs`, etc., via the `AddThis(G4Box&)` methods, etc., which can be implemented for the graphics system (but is not mandatory). Otherwise, the base class implementation demands another representation of the object (such as a `G4Polyhedron` or `G4NURBS`) which becomes appended to the graphics system scene via the `AddPrimitive(G4Polyhedron&)`, etc. methods, which *every* graphics system scene must implement (see figure 5.6).

5.4.3 The `G4VView` base class

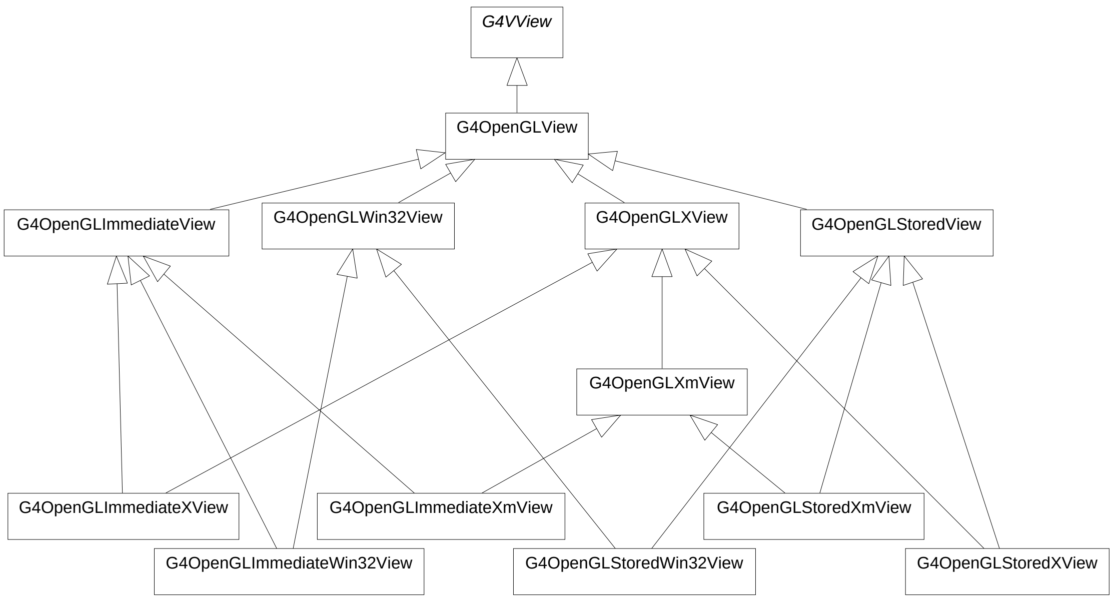

Figure 5.7: The OpenGL implementation of `G4VView`

`G4VView` requires just three mandatory `virtual` methods (and offers other non-mandatory `virtual` methods) in any derived classes that are to be non abstract (see figure 5.8). The methods are those required to set (i.e., to get light positions, calculate the projection matrix, etc.), clear (i.e., to reset graphics system states, erase the contents of all buffers, etc.), and draw a view. For the benefit of graphics systems that require separate notification at the end of the modeling phase to perform some function (e.g., the Fukui Renderer needs to have the whole model before it can begin processing it for hidden line removal), there are declared two further methods which need not be supplied by all view implementation classes.

`G4VView` keeps a record of viewing parameters requested by the user (e.g., wireframe or solid, polyhedron or NURBS). It is up to the graphics system implementation classes to decide when the geometry database (kernel) needs to be visited. This may be necessary if the user changes one of the viewing parameters, or if a new model is specified. The decision is left with the graphics system implementation classes, as some graphics systems (e.g., OpenGL in stored mode) are able to utilize a stored list of their proprietary graphics primitives (e.g., display lists (see chapter 3)) which need not be recompiled if the user wishes simply to view the scene from a different angle. Other graphics systems (e.g., OpenGL in immediate mode) need the geometry hierarchy described to them each time a view is updated.

```
virtual void SetView () = 0;
virtual void ClearView () = 0;
virtual void DrawView () = 0;
virtual void ShowView ();
virtual void FinishView ();
```

Figure 5.8: `virtual` function declarations of `G4VView`

5.5 The OpenGL implementation in Geant4

In implementing the OpenGL graphics library within the visualization framework of Geant4, every attempt has been made to utilize the paradigms of object orientation. The implementation does not realize OpenGL as a single graphics system, rather (currently) as six. As shown in figure 5.3, the six are the combinations of stored or immediate mode OpenGL interfaced to XWindows (G4OpenGLX), XWindows with the Motif toolkit (G4OpenGLXm), and the 32bit operating system Microsoft Windows variants — Windows98 and NT4.0 (G4OpenGLWin32). This is achieved partly through multiple inheritance in the classes derived from `G4VView` (see figure 5.7). Multiple inheritance has enabled the encapsulation of 'storedness' and 'immediateness', as well as the flavour of windowing system. Indeed, to further promote code reuse, the Xm implementation is inherited from (or extends) the XWindows implementation. Below is a description of the different flavours of OpenGL implementation within OpenGL.

5.6 The immediate mode implementation

The currently implemented immediate mode OpenGL Geant4 graphics systems `G4OpenGLImmediateX`, `G4OpenGLImmediateXm`, and `G4OpenGLImmediateWin32` are each able to create an instance of a `G4OpenGLImmediateScene`, and an instance of a view relevant to the graphics system. The excerpt from the `G4OpenGLImmediateX` header below is typical of the amount of functionality required of a Geant4 OpenGL `G4VGraphicsSystem` implementation class.

```
class G4OpenGLImmediateX: public G4VGraphicsSystem {
public:
G4OpenGLImmediateX ();
G4VScene* CreateScene ();
G4VView* CreateView (G4VScene&);
};
```

Figure 5.9: A typical declaration of the `G4VGraphicsSystem` base class.

5.6.1 The immediate mode scene

Most of the functionality of the scene implementation classes is within `G4OpenGLScene`, from which `G4OpenGLImmediateScene` inherits. `G4OpenGLScene` is described later. The immediate mode scene keeps a record (via a static data member) of the number of `G4OpenGLImmediateScene`s there are, for identification purposes, and so the user can refer to them individually.

5.6.2 The immediate mode view

When an OpenGL immediate mode view is requested, it is assumed that the user wishes to avoid the use of display lists. A user might be motivated to do so if their server might soon baulk at the memory requirements of a display list implementation (although protection is given against this - see 5.7) or if they are debugging a large (and maybe slow to render) geometry. The reason that the latter may be a motivation for the use of immediate mode is that the implementation of immediate mode OpenGL in Geant4 requests the use of a single buffered visual. If only a double buffered visual is available, then all drawing takes place in the front (visible) buffer, hence the progress of drawing can be seen by the user as requests are processed by OpenGL. The user who wishes to animate their view with OpenGL, will choose a stored mode view.

5.7 The stored mode implementation

`G4OpenGLStoredXView`, `G4OpenGLStoredXmView`, and `G4OpenGLStoredWin32View` comprise the currently implemented OpenGL stored mode graphics systems in Geant4. As with the immediate mode implementations of `G4VGraphicsSystem`, their sole role is to create instances of scenes and views of an appropriate nature.

5.7.1 The stored mode scene

The stored mode scene has all the functionality of the immediate mode scene (via its inheritance from `G4OpenGLScene`, but also is able to create and manage OpenGL display lists.

Display list allocation protection

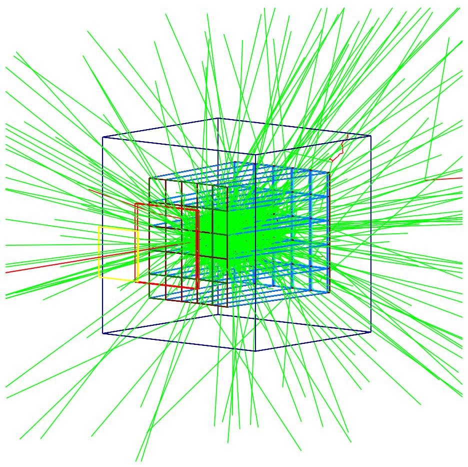

Figure 5.10: A view of an OpenGL rendered Geant4 simulation of a large electromagnetic shower in the BaBar EMC testbeam. 7000 display lists were used (in a Mesa implementation) before display list memory was exhausted. Most were used to help visualize individual simulation steps.

Upon the creation of a `G4OpenGLStoredScene`, a boolean data member of the object, `fMemoryForDisplayLists`, is set (initially to true) to say whether there is enough memory to allocate a new display list. Each time a new primitive is added to the stored mode scene, an attempt is made to pre-allocate a new display list identifier, and set `fMemoryForDisplayLists` accordingly. If the attempt is successful, then the `G4OpenGLStoredScene::BeginPrimitives` method appends the display list identifier and associated transformation of the Geant4 primitive to two template list objects (`fPODLList`[4] and `fPODLTransformList`) and makes a call to `glNewList` to signify that all following OpenGL function calls (until `glEndList` is encountered) should be placed within the pre-allocated display list.

If the attempt to allocate a new display list identifier fails, then a warning message is issued to the user, `fMemoryForDisplayLists` is set to false, and all subsequent drawing proceeds in an identical fashion to immediate mode, i.e. to the front buffer, without the use of display lists. If the scene is cleared (and display lists destroyed) then another attempt is made to allocate a display list id., and `fMemoryForDisplayLists` is set accordingly to potentially reenable stored mode operation.

Persistent and transient geometrical objects

The Geant4 primitives whose representations in OpenGL may be placed inside a display list fall into two catagories; *persistent* and *transient*. Persistent objects are deemed to be those with run duration persistence, such as items of detector, whereas transient objects are deemed to be those with less than run duration persistence, such as steps[5]. Either category of Geant4 primitive may be placed inside a display list, although they are treated differently.

When a persistent geometrical object is placed in a display list, it only becomes rendered in the view at the end of modeling, when the display list is executed in a `glCallList` function call. Typically, all the display lists which are representative of persistent geometry elements in the current view are called from a single super display list (`fTopPODL`) which may then be executed with a single call to `glCallList`. However, transient objects are representative only of artefacts of simulation (i.e., steps, hits, etc.) and may be very numerous, and produced at every

[4]PODL stands for Persistent Object Display List.

[5]A step is two adjacent points in a tracked particle's trajectory, between which the particle undergoes some change.

step of the simulation. For these reasons, drawing of transient objects (in or out of display lists) proceeds in the front framebuffer. In addition, transient objects which are placed in display lists are executed (rendered) at the same time as being compiled into the display list, so that the user may see the progress of the simulation (which may take some considerable time). No effort is made to make a super display list for transient geometrical objects on account of their transient nature. For the duration of their existence, their display list identifiers and associated transformations are kept in two lists (currently, both are objects of a RogueWave template class, `RWTValOrderedVector<class>`), the members of which are executed recursively when required.

Display list reuse

In an attempt to reduce the amount of redundancy in the allocation and compilation of display lists, the `G4OpenGLStoredScene` maintains a dictionary of Geant4 primitives it already has a display list for. The dictionary is composed of `G4VSolidPointer` objects (a typedef of `unsigned int`, and able to represent any pointer value, as well as having a well defined method for comparison) and each entry has associated with it a display list identifier. The dictionary becomes useful if a detector is composed of many repetitions of a single `G4VSolid` with different transformations. By storing the display list pertaining to each element of the scene and its transformation separately (the `PODLList` and `PODLTransformList`), the stored scene is able to decouple a Geant4 primitive from its current transformation, and so reuse the display list information for many orientations of the primitive.

5.7.2 The stored mode view

The stored mode OpenGL view attempts to render via the use of display lists, and by that token, it must know when those display lists become insufficient to describe the scene as the user has requested (e.g., if the user has changed the material density below which volumes are culled[6]). `G4OpenGLStoredView` (from which all stored mode implementations of an OpenGL view inherit - see figure 5.7) contains the methods to compare the current requested viewing parameters against those with which the current display lists were constructed, and so is able to trigger a Geant4 geometry kernel revisit if necessary.

It is the job of the windowing system specific Geant4 view implementation (e.g. `G4OpenGLXView`) to make the connection between OpenGL and the windowing system, as well as appropriating the necessary resources (e.g., colormaps, visuals, windows, etc.) for OpenGL to be able to use the windowing system effectively. Motif being an extension of XWindows has enabled the Geant4 OpenGL Motif implementation to be derived from the Geant4 OpenGL XWindows implementation. With a sensible division of tasks, methods have been written in the XWindows implementation that can be used by the Motif implementation (e.g., for creation of colormaps, definition of viewports, and synchronisation of the connection to XWindows from OpenGL).

5.8 Features of all OpenGL implementations in Geant4

Each of the six graphics system implentations of OpenGL in Geant4 has their own set of features and reasons why a user might request them. However, all OpenGL specific functionality has purposely been placed inside windowing system independent classes, so that subsequent graphics system implementations of OpenGL may inherit from them and share their code. A summary and brief description of the features is given below.

[6]Culling is a process whereby data are ignored if they fail to meet certain criteria.

5.8.1 Polyhedron representation

The OpenGL graphics library is unable to make use of most of the Geant4 primitives 'as is', and so when they become appended to an OpenGL scene, by default, the drivers request the kernel to supply polyhedron representations for each primitive. Each primitive is split into three and four sided facets before the vertices comprising these facets are communicated to the OpenGL drivers. The OpenGL state machine is prepared to expect quadrilaterals[7], necessitating the duplication of the last vertex for each facet for which only three vertices have been specified — the kernel guarantees to never attempt to supply a facet with more than four or less than two sides. A boolean flag is passed with each vertex from `G4Polyhedron` to describe whether the line linking the current vertex to the next vertex is physically representative of a some portion of the volume (or is an approximation to such) or is just an artifact of the decomposition of the volume into a polygon. This flag may be used by OpenGL to control the rendering of individual lines in a polyhedron. For example in drawing a wireframe cylinder (which by default in Geant4, is represented by 48 triangles (two 24 segment circles) and 24 rectangles) the 24 bars joining the two circles and the 'spokes' in each of the ends of the cylinder are not necessary to represent it, and so are not drawn.

Once the vertex information has been communicated to the OpenGL scene, some processing in respect of the user's specified visualization attributes is possible. Some of the available options and their execution using OpenGL are described below.

5.8.2 Wireframe

The representation of solids as wireframes in OpenGL requires the least of the implementation's vertex processing abilities. To increase performance, lighting and

[7]OpenGL is often able to optimize the primitive assembly phase of modeling if it knows that all supplied primitives will be of a given type (e.g., quadrilaterals, triangles, etc.)

back-face culling[8] are disabled, although depth testing is left enabled to preserve depth sense if wireframe objects are drawn in the same scene as filled-face solid objects.

5.8.3 Hidden line/surface removal

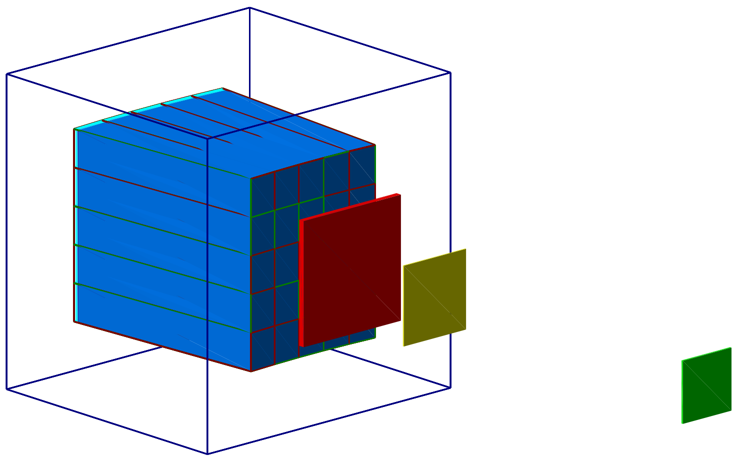

Figure 5.11: A view of the PSI BaBar EMC testbeam with hidden surface removal enabled

[8]Back-face culling is a rendering optimization, whereby facets with their surface normals pointing away from the eye point are discarded (as they would be obscured by foreground facets anyway). The technique can reduce by a half, the number of facets to be rendered.

Hidden surfaces

Hidden line/surface removal requires some degree of extra processing over simple wireframe drawing. A scene with the hidden surfaces removed can be easily achieved with OpenGL, by requesting that polygons be drawn with their interiors filled, and that depth testing is enabled. To add realism, lighting is also enabled, and for speed, back-face culling is enabled. The filled faces of the polygons are given ambient and diffuse material properties derived from their constituent vertex colours, from which light may reflect. The depth testing ensures that the surfaces which remain after back-face culling, are not obscured by any surfaces which pass behind them.

Hidden lines

The hidden lines in a wireframe view of a model are those which would be obscured by front facing facets if the model was rendered as a collection of solid objects. The process of removing hidden lines is meant to provide a simple view of surface features of the model, without introducing the additional colour information associated with the filled facets. There are many algorithms for the removal of hidden lines, but most rely on the complete scene being rendered at least twice. Currently, hidden lines are removed in the Geant4 OpenGL drivers, by drawing each facet of the model as a wireframe followed by a filled polygon. The filled polygon is drawn in the current background colour of the view, and obscures any lines it passes over. Depth buffering ensures that lines drawn on the visible surface of the model do not become obscured. This method of hidden line removal suits the way that information is made available to the OpenGL drivers, in the sense that the whole model need only be traversed once, and need not be stored. Many alternative techniques require the model to be drawn in full, at least twice. One possible disadvantage to the implemented technique is its inclusion of depth buffering artifacts. These express themselves as discontinuities of lines, where the filled polygon drawing occasionally mistakenly obscures lines where the depth test occurs between two very similar depth values. Such conflicts in the depth buffer are easily overcome by the use of polygon offsetting, a feature of version 1.1 and later of the OpenGL API, which enables some factor and/or a bias to be applied to all vertices being representative of a particular form of polygon (filled or wireframe) which are being compared in the depth buffer.

5.8.4 Haloing

Haloing is a procedure which may be applied to a wireframe drawing, whereby lines in the foreground are drawn complete, and lines passing behind them in the background, are interrupted for some distance either side of a foreground line. This helps to distinguish between foreground and background in a wireframe drawing, and also reduces the amount of visual information in a scene that may be very cluttered with lots of detail. The reduction of information comes at no expense to objects in the foreground.

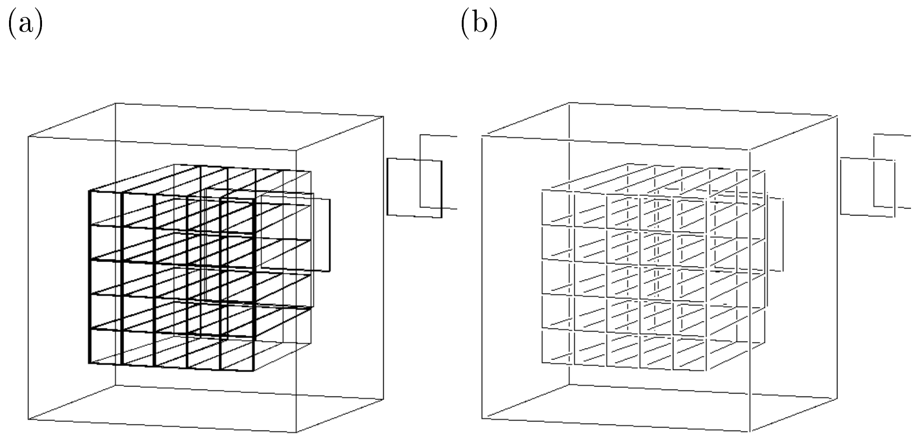

Figure 5.12: (a) A wireframe visualization with haloing disabled. (b) A wireframe visualization with haloing enabled.

To achieve haloing, two rendering passes must be completed. First, the wireframe objects must be rendered with a thick line width, into the depth buffer alone. This creates a chunky image of the wireframe drawing in depth coordinates. Next, the scene is rendered into the colour buffer, using a thin line width and depth testing. The depth testing algorithm is set so as only to permit the drawing of pixels into the colour buffer where the pixel depth value is equal to that stored in the depth buffer for that pixel position. As the depth buffer contains a chunky image of the 'foreground mask' of the wireframe scene, all the foreground lines get drawn, and where they cross with background lines, an exclusion zone exists around the foreground line to make it clear which passes in front of which.

5.8.5 Encapsulated PostScript file production

The ability to generate hard copy output of the results of Geant4 visualization is of secondary importance to the OpenGL drivers, as another element of the toolkit, DAWN[9] [8], is able to produce very high quality PostScript files. However, quality hardcopy output is a desirable feature of any graphics system, and so some solution in OpenGL was sought.

The pixmap solution

The first solution to hardcopy output investigated was to simply produce an 'automated screen dump' of the rendered output of OpenGL. OpenGL has the concept of a rendering context, to which it sends the output of its rendering phase. If rendering is intended to update the framebuffer, then the rendering context is associated with an area of the screen. Alternatively, the rendering context can be defined as an area of memory. The rendering phase is where the OpenGL model becomes decomposed into fragments via the application of the projection transformation matrix and defi-

[9]Drawer for Academic WritiNgs, written by Satoshi Tanaka et al., Fukui University, Japan.

nition of a rectangular viewport (see Chapter 3). As each fragment is a collection of data necessary to define the contents of one pixel in the rendering context, it is this decomposition into fragments that imposes a certain image resolution on the final output. The pixmap solution to hardcopy output by the Geant4 OpenGL drivers defines a rectangular array of objects in memory, each of which is able to accommodate one fragment. OpenGL is then instructed to render to this context, after which the array contains fragment information relevant to the OpenGL model. The RGB values (or intensity value, if a greyscale picture is required) of each fragment in the array is then written to a file. This file has a PostScript header to instruct any device reading the file, to interpret the data as RGB pixels (or greyscale pixels).

The pixmap approach to producing hardcopy of Geant4 OpenGL visualization produces much the same sort of result as may be obtained by the user 'grabbing' the contents of a graphics window by hand (using xv or xwd for example). In addition to the image being produced at a resolution defined by the dimensions of the viewport to which the model was rendered (which may well not be as fine as the resolution offered by most laser printers) the files produced can be very large, especially if a colour image is described (as this requires three integer values to be supplied for every pixel in the image).

The vectored PostScript solution

Many of the disadvantages of the pixmap solution to hardcopy output of OpenGL visualization stem from the fact that the rasterization phase introduces a finite image resolution. Fortunately, OpenGL can be placed in a 'feedback' mode of operation (via the command `glRenderMode (GL_FEEDBACK);`) whereby OpenGL processes the model as far as the rasterization phase, and then makes available for inspection the geometrical primitives it has produced as being representative of the model (see section 3.6.3). By stopping short of the OpenGL rasterization phase, an opportunity

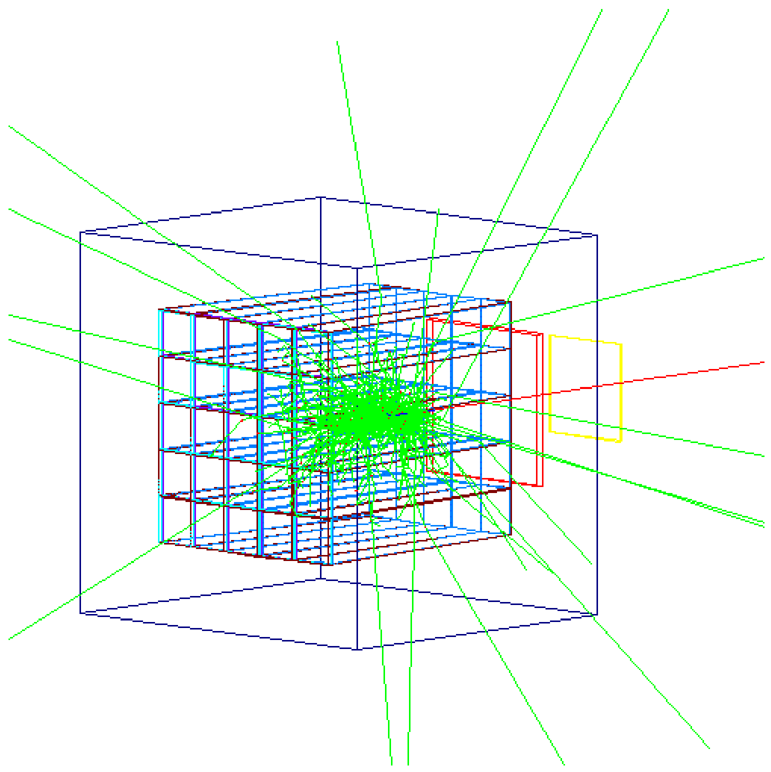

Figure 5.13: A pixmap OpenGL wireframe representation of the PSI EMC testbeam with the steps produced in an electromagnetic shower shown.

is given for some other 'rendering engine' to process the work done by OpenGL in its modeling phase and subsequent transformation into NDCS space. It is possible, as was first demonstrated by Mark Kilgard [13], to generate a list of triangles from the planar primitives returned by the OpenGL modeling and transformation phases. Given a PostScript description of an arbitrary triangle such as that provided by Frederic Delhoume [4], a file can be written that describes the OpenGL model as a set of device resolution independent triangles. These techniques have been incorporated into the Geant4 OpenGL drivers, to offer the Geant4 user a high quality, vectored PostScript output option.

A disadvantage of the vectored output solution, is that it omits any of the processing steps performed in the normal OpenGL rasterization phase. One such processing step is depth testing, where the depth value of a fragment is inspected before it is allowed to update the framebuffer. To overcome this disadvantage, the list of triangles returned from the modeling and transformation phases is sorted into descending depth order (a process often referred to as the painter's algorithm) before being written to the file, so that when read by a PostScript interpreter, the first triangles drawn (and hence with the greatest potential for being obscured by subsequent triangles) are those furthest from the near face of the viewing volume. Another problem is that some operations performed on specific buffers (such as the depth buffer) which result in a discrimination over what gets drawn into the framebuffer (as occurs with haloing, for example) do not necessarily have the same effect. If a Geant4 OpenGL haloing-enabled image is saved as a vectored PostScript file, then twice as many triangles are evident in the final PostScript file as are required, as two models have been described to the PostScript rendering engine. The first describes the 'chunky' wireframe model which is only rasterized to the depth buffer when OpenGL rasterization takes place, and the second describes the thinner wireframe model which is normally rasterized into the framebuffer.

For comparison, examples of the same OpenGL visualized model printed using colour/greyscale, pixmap/vectored PostScript are given in figure 5.15. The file sizes are also given in the caption. Note that the greyscale vectored PostScript file is actually larger than the colour vectored PostScript file (but still smaller than either of the pixmap files) because, in this implementation, greyscale is achieved by forming the average of the red, green and blue values for each vertex, and then specifying this value for all the colour components (R, G, and B) for that vertex. Hence each R, G, and B value is more often fractional rather than integer, and so when written as an ASCII file, takes a few more characters to represent.[10]

[10]For example, R=1, G=1, B=0 might take as little as 5 characters (0 0 1) to represent in an ASCII file, whereas the averages, R=0.66667, G=0.66667, B=0.66667, may take just under 7 times as many characters to represent, if they were quoted to five decimal places. The truncation of floating point numbers is an operating system dependent feature.

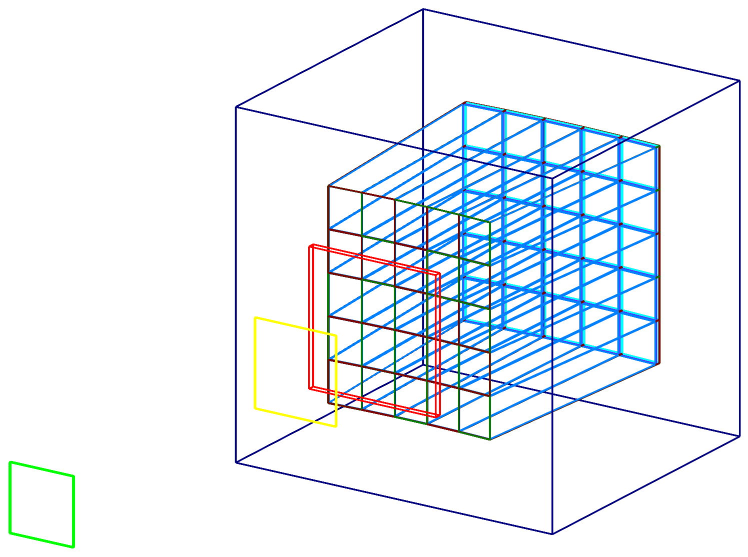

Figure 5.14: A vectored PostScript OpenGL wireframe representation of the PSI EMC testbeam.

5.8.6 The G4Xt singleton class

The OpenGL drivers interfaced to Motif provide an event driven visualization tool for the Geant4 user. The events (in the XWindows sense) are generated by the interaction of the user with the application. The usual way of creating an *event aware* application, is to allow the application to inspect all XWindows events, and respond to those which are of interest. In the case of the OpenGLXm drivers for Geant4, the Geant4 kernel is the true application, and the OpenGLXm drivers are just clients of it, so there is a need for an XWindows event getting loop to be a constituent part of the Geant4 kernel. This component must have responsibility for catching all XWindows events that are of interest to the Geant4 application, and then distribute them to the component of the toolkit to which they are relevent. The task is performed by the `G4Xt` class, designed by Guy Barrand [3] of LAL, Orsay, and allows many XWindows event-aware components of the Geant4 toolkit (e.g. GAG, Momo, OpenGLXm, OPACS) to coexist, without 'locking out' each other (as might otherwise happen if a single component contained an event getting loop which responded only to those events of interest to itself, and discarded all others.)

(a) (b)

(c) (d)

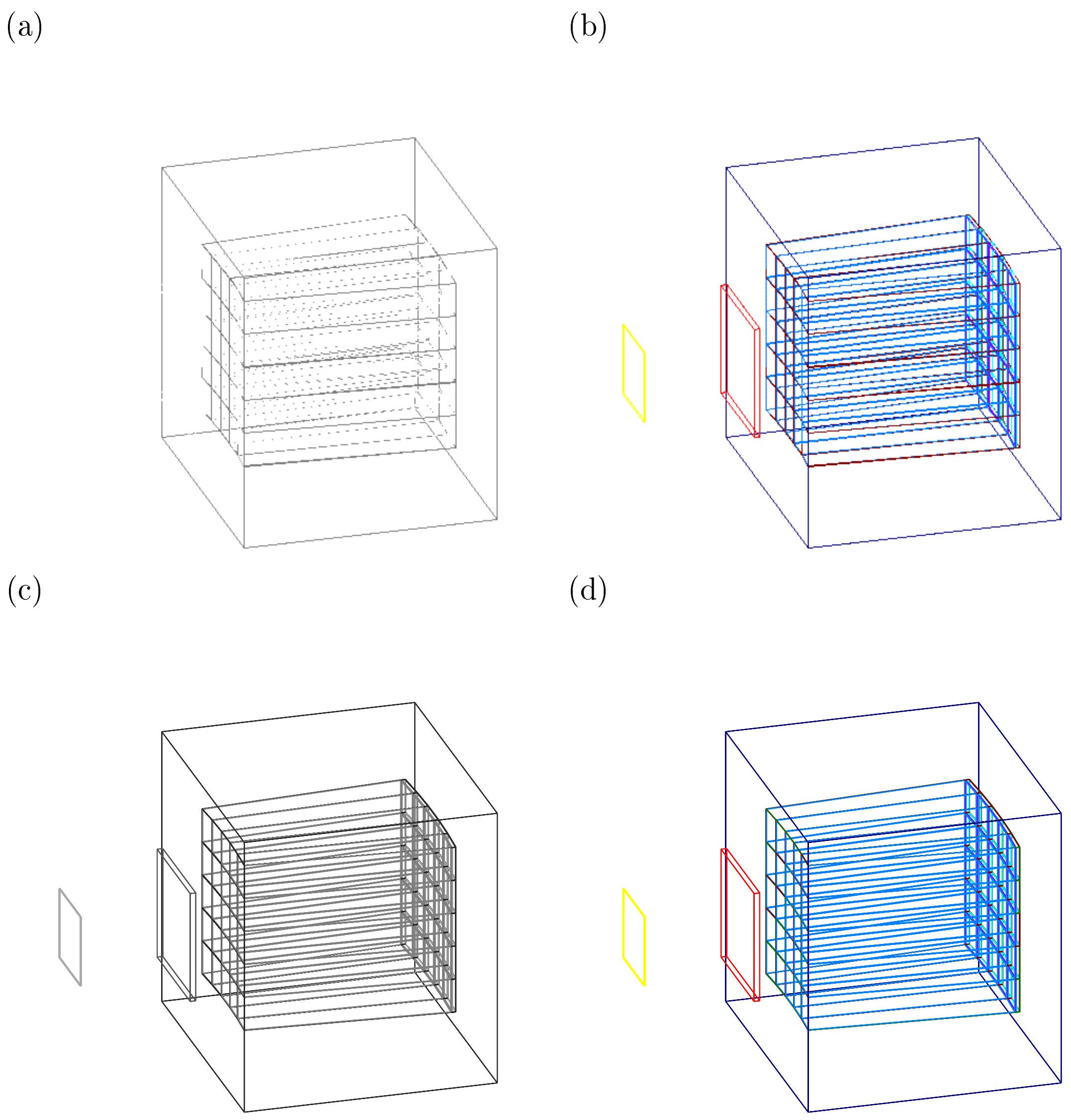

Figure 5.15: Four Encapsulated PostScript images of the BaBar EMC Testbeam at PSI (file size in brackets). (a) Greyscale pixmap (758476 bytes) (b) Colour pixmap (2274103 bytes) (c) Greyscale vectored PostScript (337887 bytes) (d) Colour vectored PostScript (281943 bytes)

Chapter 6

Parameterization in Geant4

6.1 Introduction

The role of the simulation elements of the Geant4 detector simulation and visualization toolkit are to produce a faithful likeness to data that may be produced by the real experiment that is modeled in Geant4. The motivation for such simulation is often to produce a larger number of certain types of event (or events in a certain part of the detector) so that a better feel for what to expect from the real experiment may be gained.

In a detailed simulation, the trajectory followed by a particle is approximated to a succession of steps. Before each step is taken, a decision is made regarding how the step will be limited in the interaction of the particle with its current environment — which includes the material it is currently traversing, any electromagnetic fields present (if the particle has electric charge), and the lifetime of the particle compared with its current age — and at the end of the step, the particle state is modified accordingly. Any energy lost by the particle is recorded as a *hit* in the current sensitive detector (if one has been declared). At the end of each event, the hits stored in each sensitive detector may be combined into a *digitisation*. Digitisations are of a similar form to the real data read from an experiment, and may enter the *reconstruction* phase, where all detector signals (or digitisations) are inspected and macroscopic features extracted accordingly (such as clusters, helices, extracted photon energies, etc.)

A typical Geant4 simulation of 100 MeV electrons passing through Caesium Iodide results in 220.7 ± 17.3 hits per initial particle. Although the detailed simulation is thorough and (hopefully) faithful, it is very CPU intensive and time consuming, and so may not suit the needs of a user wishing to generate a very large pool of data. A *parameterized* simulation is one for which a detailed study of certain events in a certain medium has been undertaken, and a 'shortcut' produced on the grounds of this. The shortcut may produce hits (as does the full simulation) or digitisations, or any other products of simulation, but must remain faithful to the detailed simulation, within some tolerance.

6.2 Parameterizations code in Geant4

Geant4 provides a framework into which may be placed a user's parameterization. Essentially, the user defines a parameterization envelope — which may be specific volume elements of a detector, or some other artificial area — in which the user wishes the parameterization to become applied. The user also must write a class that inherits from `G4VFastSimulationModel`. This class should define three `virtual` methods of `G4VFastSimulationModel` (two of which are mandatory) which are intended i) to determine the species of particles whose entry into the parameterization envelope is sufficient to trigger the parameterization, ii) to trigger the parameterization if the particle state falls within the acceptable range of particle states (e.g., particle energy is low enough, and it is far enough away from a boundary), and iii) to perform the actions and processes defined by the parameterization, which may include 'killing' the incident particle, producing some detector response, and propagating some (lesser energy) particles out of the other side of the parameterization envelope. The parameterization envelope is made known to the derived class of `G4VFastSimulationModel` when an instance of the derived class is made.

```
class G4VFastSimulationModel {
public:
G4VFastSimulationModel(const G4String&);
G4VFastSimulationModel(const G4String&, G4Envelope*, G4bool);
G4VFastSimulationModel() {};

virtual G4bool IsApplicable(const G4ParticleDefinition&) {
return true;
}
virtual G4bool ModelTrigger(const G4FastTrack &) = 0;
virtual void DoIt(const G4FastTrack&, G4FastStep&) = 0;
...
};
```

The framework into which a parameterization model must fit was designed by Marc Verderi and Paolo Mora deFreitas of *ÉcolePolytechnique*, and receives a fuller discussion in the Geant4 User's Guide for Application Developers [2].

Chapter 7

Electromagnetic Shower Parameterization

7.1 Approaches to parameterization

In terms of speed, the performance of a parameterization depends on the mechanism by which the parameterization gets called, the algorithms implemented to describe the processes being parameterized, and the amount of processing which the product of the parameterization requires before can it be compared to real data (i.e., forming digitisations out of hits, clusters out of digitisations, etc.). The issue of how the parameterization algorithm gets called is fixed for Geant4 in the framework (see Chapter 6), but the issues of algorithm efficiency and how readily digitisations (to be compared with real data) can be formed from the parameterization output may be chosen by the implementor.

There are many ways in which patterns may be sought in data such that a parameterization may be produced. This chapter describes some preliminary studies of the description of electromagnetic showers in CsI(Tl) using data simulated by the Geant4 toolkit. These studies were concerned only with the possibility of producing simple and very fast algorithms. The ability to 'kill' a tracked particle whose energy and distance to travel in the current sensitive volume have reached some user defined threshold already exists in the Geant4 toolkit. The studies were aimed at progressing beyond this simple energy and range dependent 'cut' approach.

Some data relating to the model used in the Geant4 simulation are given in appendix B.

7.2 Backscattering of shower energy

When $e^{\pm}$, γ are incident upon CsI(Tl), some of the ensuing electromagnetic shower may be directed back out of the surface of incidence. In the case of $e^{\pm}$, γ entering a sensitive calorimeter element from an insensitive volume such as the atmosphere then that portion of the shower which is scattered behind the plane of incidence is deemed unable to deposit energy in the sensitive volume. Hence any attempt to parameterize electromagnetic showers due to incident $e^{\pm}$, γ in CsI(Tl) must account for this backscattering effect. The reduction of total shower energy as a function of normally incident particle energy is given for $e^{\pm}$, γ in figure 7.1.

7.3 Profiles of electromagnetic energy deposition in CsI(Tl)

Two studies relating to electromagnetic shower development in CsI(Tl) were undertaken in an attempt to highlight a profitable course in the pursuit of a parameterization. The first of these studies was an attempt to quantify in some way how the 'containment' of electromagnetic showers initiated by normally incident $e^{\pm}$, γ in CsI(Tl) varied with incident particle energy. Of course, it is difficult to try and quantify the containment of an electromagnetic shower in matter, unless contain-

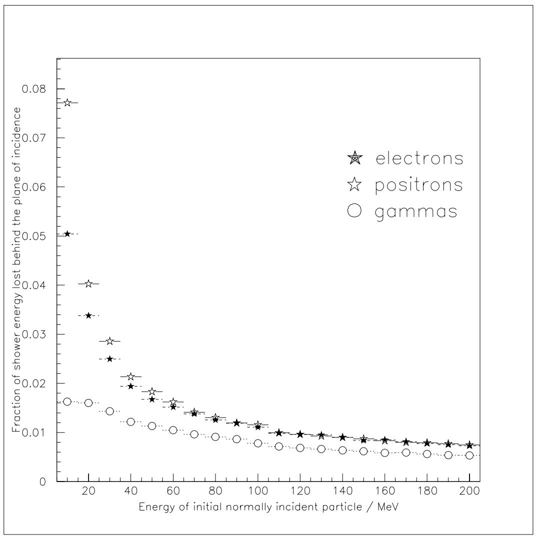

Figure 7.1: The fraction of an electromagnetic shower's energy which is back scattered behind the surface of incidence for normal incidence $e^{\pm}$ and γ initiated electromagnetic showers in CsI(Tl)

ment relates to the *average* containment of *some fraction* of the total shower energy.

7.4 Envelopes of 99% shower containment

The radiation length[1] for CsI(Tl) is 1.85cm and provides us with a suitable scale for the exponential decay of electron energy with distance travelled. For the purposes of the study it was decided to record the containment of 99% of shower energy in CsI(Tl). At each incident particle energy studied 64,000 electromagnetic showers were simulated and the Geant4 hits produced by these were binned according to how far into the CsI(Tl) test volume (see appendix B) they had penetrated[2]. Each

[1]The radiation length (X_0) in a material is defined as the mean distance over which a high energy electron ($E \geq\sim 20MeV$) loses all but $1/e$ of its energy by bremsstrahlung.

[2]Penetration was defined in these studies as the distance from the surface of incidence upon the CsI(Tl) volume along the incident trajectory.

bin covered a penetration of 12mm. Once the bins had been filled, their contents were sorted into ascending order of perpendicular displacement from the incident particle axis, and the 99th percentile for each bin was sought. The results are shown in figure 7.2.

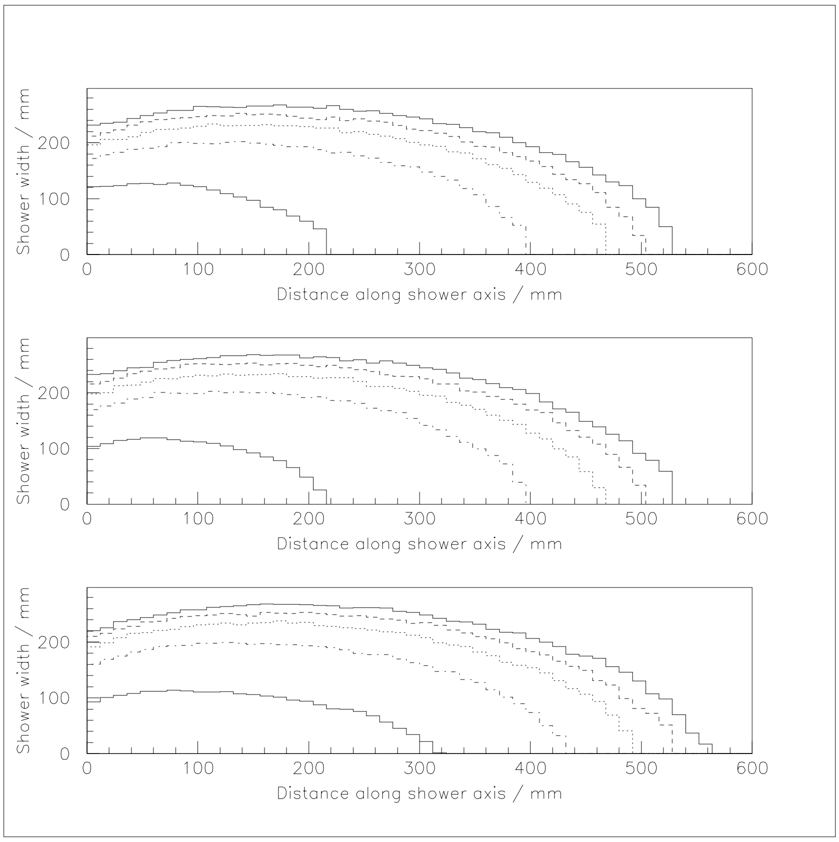

Figure 7.2: Envelopes of 99% shower containment for electromagnetic showers induced by e^+ (top) e^- (middle) and γ (bottom) in CsI(Tl). The five envelopes correspond in each case to showers induced by initial particles of energy 10 MeV (innermost envelope), 50 MeV, 100 MeV, 150 MeV, and 200 MeV (outermost envelope).

7.5 The distribution of energy deposition in electromagnetic showers

It is known that for electromagnetic showers in a particular medium approximately 10% of the total shower energy lies outside a cylinder of radius R_M[3], centred on the incident particle trajectory's axis and approximately 1% of the energy lies outside the cylinder of radius 3.5 R_M [7]. It is also known that beyond a radius of 3.5 R_M and beyond, composition effects become more important and the scaling with R_M begins to fail.

The second study involved conducting a series of simulations of electromagnetic shower production in CsI(Tl) initiated by each of $e^{\pm}$, γ at various energies. For each complete shower simulation the hits generated were sorted into ascending order of perpendicular distance from the incident particle axis. This list was then traversed, and the perpendicular distance by which 10%, 20%, ..., 100% of the total energy contained in that shower was deposition was recorded. When the percentile data for all simulated showers initiated by a given particle species of a given initial energy were combined they produced the average distribution of energy deposition across showers of that origin. This distribution was produced to reveal whether the distribution of energy depositions within an electromagnetic shower develops in a way that can be reconciled with the development of shower containment profiles given in figure 7.2.

The data produced in the second study are shown in figure 7.3.

[3]R_M is the Molière radius. For CsI(Tl), $R_M = 3.8$cm.

7.6 Conclusions

It was found that the average distribution of energy deposition throughout electromagnetic showers originating from $e^{\pm}$, γ becoming incident upon CsI(Tl) varied with the energy of the incident particle. The average radially symmetric envelope required to contain 99% of all shower energy was also found to vary with the energy of the incident particle.

The studies suggest that a profitable development of this form of parameterization, may be to sub-divide a parameterized shower containment envelope into a number of contiguous volumes, and then study the energy deposition into these volumes. The successful treatment of the correlation between the deposition of energy into each volume and all the others would also allow the parameterization to distance itself from the radially symmetric nature of the *average* electromagnetic shower in CsI(Tl).

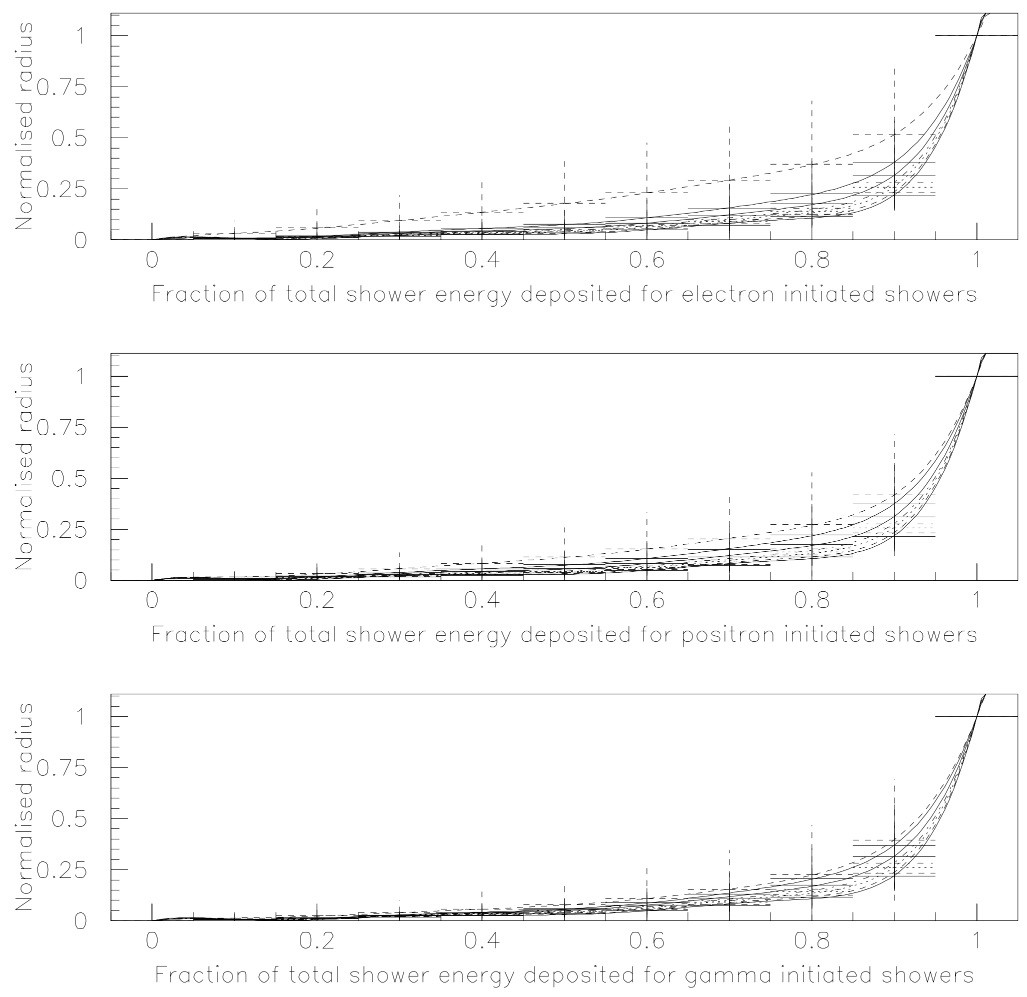

Figure 7.3: The distribution of energy deposition in electromagnetic showers induced in CsI(Tl) by e^- (top) e^+ (middle) and γ (bottom). The data relate to showers of energy 10 MeV (uppermost curve), 50 MeV, 100 MeV, 150 MeV, 200 MeV, 300 MeV, and 400 MeV. For each hit (energy deposition) contributing to the plots the radius was normalised to the maximum radial extent for each shower.

Chapter 8

Results

8.1 Introduction

The design and implementation of the OpenGL drivers form the major part of the work for this thesis. The Motif extensions offer facilities over and above the basic Geant4 interface — the Geant4 visualization philosophy of "value adding". We summarize the acheivements in the next section — details can be found in earlier chapters.

The work also includes a significant investigation into the parameterization of electromagnetic showers for particle physics detection with particular relevance to the BaBar experiment. This experiment will be the first to use Geant4 and its new facilities. The results of the investigation are also summarized in this chapter.

8.2 The Geant4 OpenGL drivers

The Geant4 user requirements document [1] laid out a minimal set of requirements that the visualization components of Geant4 should satisfy. The Geant4 OpenGL drivers are able to satisfy all those requirements for which there exists an abstract visualization interface (see table 5.1).

The current capabilities of the Geant4 OpenGL drivers are outlined below for the drivers interfaced to XWindows and Motif.

8.3 Features available in OpenGLX drivers

Feature	Options
CSG[1]drawing	- wireframe - hidden lines removal - hidden surfaces removal - hidden lines and surfaces removal
NURBS[2]rendering	- wireframe - filled polygon
Rendering mode	- immediate - stored
Viewer operations[3]	- pan up/dow, left/right - rotate - slice scene (DCUT in Geant3)
XWindow operations	- user resize by drag - auto update on expose

Table 8.1: Visualization features available via the OpenGLX drivers.

[1]Constructive Solid Geometry

[2]Non-Uniform Rational B-Spline

[3]Viewer operations as defined by the Geant4 visualization interface

8.4 Features available in OpenGLXm drivers

The visualization features currently available via the OpenGL drivers interfaced to the Motif toolkit, are given in table 8.2.

Feature	Options
Interactive control	- pan up/down, left/right - orbit - zoom - dolly - orthographic/perspective view - transform scene/camera - drawing style - rendering style
Hardcopy output	- Rasterized/vectorized PostScript - Greyscale/colour file production
Antialiasing of lines	- on - off
Transparency of solids	- on - off
Haloing of wireframe objects	- on - off
Background colour	- black - white

Table 8.2: Visualization features available via the OpenGLXm drivers.

A typical example of the presentation of the Geant4 OpenGLXm viewer is given in figure 8.1

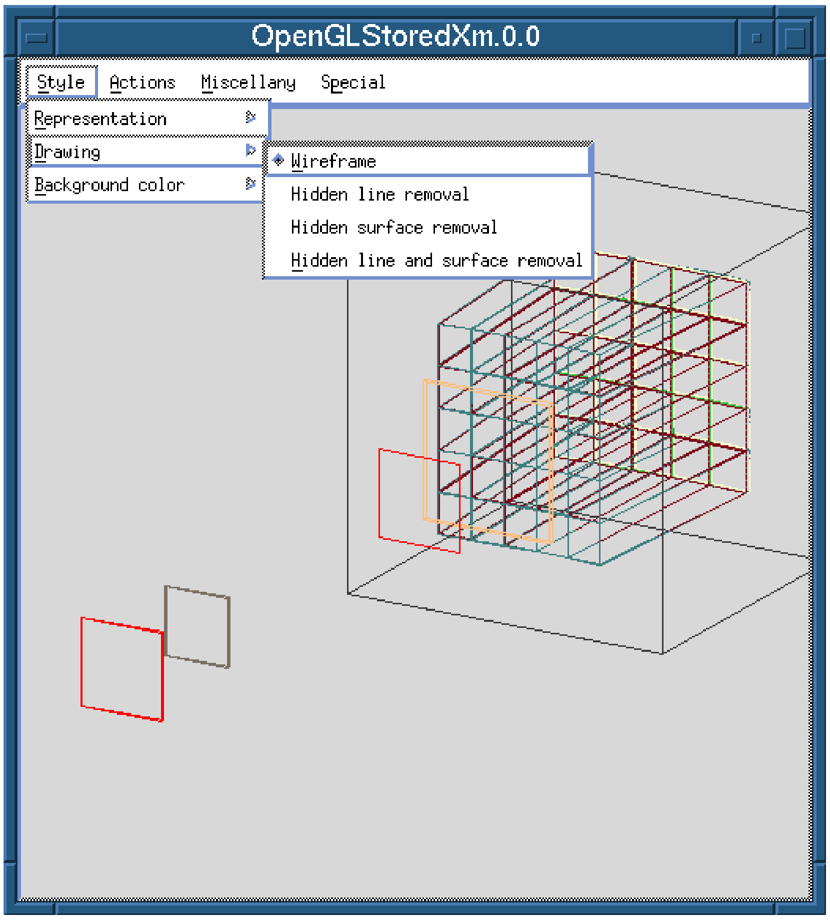

Figure 8.1: An example of an OpenGLStoredXm session.

8.5 The parameterization of electromagnetic showers in CsI(Tl)

The studies of $e^{\pm}$ and γ induced electromagnetic showers produced a consistent picture of the showering process in CsI(Tl), but also exposed the difficulty of producing a simple algorithm to intercept $e^{\pm}$ and γ which is bias free. Scope for further investigation is highlighted in section 7.6, but is beyond the scope of this thesis.

Chapter 9

Conclusions

9.1 Geant4

The Geant4 project has brought together the effort of over 100 physicists around the globe to design and produce a toolkit of components using object oriented methodologies and C++. This scale of division of labour has been possible largely through the use of object orientation, highlighting the potential value of such practices to the particle physics community.

9.2 G4OpenGL drivers

OpenGL has established itself as the foremost cross-platform graphics library, and lends itself well to the concept of desktop computing that is of increasing importance to particle physics. The employment of C++ and OO paradigms have enabled much commonality between the various Geant4 OpenGL drivers to be factored out and reused by means of inheritance promoting ease of code maintainance and a greater longevity. The Geant4 OpenGLXm implementation exploits the concept of value adding to provide functionality beyond that available in the Geant4 visualization interface, such as viewer interaction and the ability to make hardcopy output. The concept of *picking*[1] is not yet addressed by the Geant4 visualization interface, but the Geant4 OpenGL drivers may be extended as and when necessary.

The Geant4 OpenGL drivers provide a base visualization functionality that may be extended by other parts of the toolkit. For example, the capabilities of the Geant4 OpenGL drivers interfaced to Motif in respect of interactive viewing and PostScript file production may be extended by the OpenInventor and Fukui Renderer elements of the toolkit respectively.

G4OpenGLXm widget wrapper classes

As the Geant4 visualization interface evolves and new features are added, then so may this new functionality be implemented in the G4OpenGL drivers. The widget wrapper classes have been designed specifically to ease the implementation of new features into the Geant4 OpenGLXm drivers and ensure that a consistent interface is presented to the Geant4 OpenGLXm user.

9.3 EMC Parameterization

The power of Geant4 as a simulation tool was demonstrated by the study of the development of electromagnetic showers in CsI(Tl). The study suggested that a profitable route for interception of $e^{\pm}$ and γ may be to parameterize the containment of some fraction of total shower energy, and generate a pattern of energy depositions based on a set of contiguous volumes within the containment. It is essential that the correlation between these energy depositions be treated properly. The application of this method lies outside the scope of this thesis.

[1] The ability interactively to select and operate on some element of a visualized model.

Appendix A

Introduction to UML

The unified modeling language was conceived in 1994 by Grady Booch and James Rumbaugh at Rational Software Corporation and was an attempt to create a single formalism for describing program flows and relations out of the best features of the two major formalisms of the time, namely the Booch method and OMT[1] method.

The UML is able to describe both static and dynamic views of business and software models at all stages of development. The use of UML in this report extends only as far as the presentation of static class diagrams (class diagrams) and so the interpretation and meaning of some aspects of these is described briefly below. The discussion is intended as an aid to understanding rather than a tutorial on the class diagram and its uses. A fuller description of the capabilities of UML can be found in many dedicated texts including [6].

One feature of the Rational Rose CASE modeling tool is the ability to *reverse engineer* existing code. This is the process of being able to examine a piece of code and produce UML diagrams from its analysis. Although much of the power of a CASE tool is in its application at the modeling stage of software design, its

[1]Object Modeling Technique.

facility for the visualization of (complex) hand written[2] object relationships can be an invaluable tool for the dissemination of software architecture information.

A.1 The UML class diagram

A class diagram is a model of the static relationships between classes in a system. No program flows, states of objects or collaborations between objects are shown by a class diagram.

A.1.1 The representation of the class

A class in the class diagram is represented by a rectangle, which may be split into three areas to accomodate the class *name*, class *attributes*, and class *operations*. Each attribute or operation can be classified as public (+), protected (#), or private (-). The attributes (data members) are represented as the attribute name, followed by a colon and then the attribute type. The attribute type can be an intrinsic data type, or any user defined data type or class, and may be followed by a default value. An operation (member function or method) is followed by the round-bracketed list of arguments to that operation, and by a colon and return type (if the return type is non-void). The intrinsic data types available to different languages vary, so the UML designer may either avoid the use of intrinsic data types, or bind the UML design to a particular language at this stage.

Operations and attributes may be included or excluded from the class diagram when using the Rational Rose CASE tool. In order to simplify the class diagrams in this report, all class diagrams have attributes excluded, and all but the class diagram of the G4OpenGLXm widget wrappers (figure 4.7) have the operations excluded.

[2]i.e. written without the use of a CASE modeling tool.

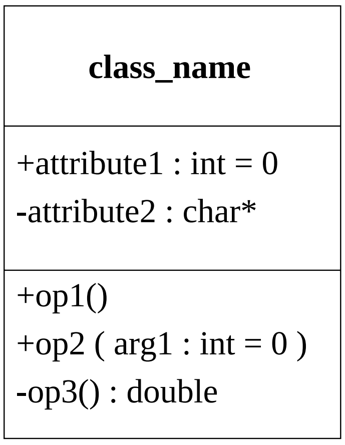

Figure A.1: The representation of the class in UML

A.1.2 Generalization

The concept of inheritance in UML is more usually referred to as *generalization* and is depicted by an arrow with a hollow closed head leading from the sub-class to the super-class (daughter to parent). Caution must be exercised with generalization if the UML design is to remain unbound to any particular language. Although C++ allows multiple inheritance (such as is used by the *G4OpenGLStoredXView* class in the Geant4 OpenGL drivers) some other OO languages (such as Java) do not (a Java class may *extends* no more than one other Java class).

A.1.3 Aggregation

Aggregation is a term used to describe the composition of some element of a model in terms of other elements of the model. There are two types of aggregation: shared aggregation and composition aggregation.

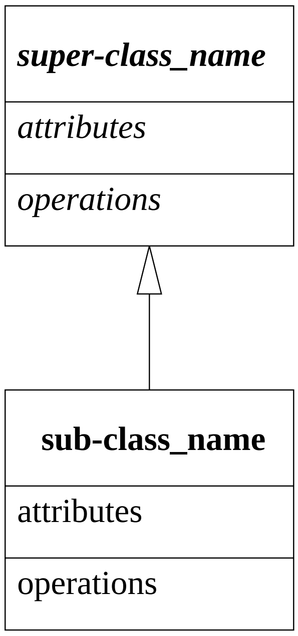

Figure A.2: Generalization in UML. The super-class is abstract.

Shared aggregation

Shared aggregation can be thought of as **a** *contains a* **b**. Shared aggregation is not exclusive, so **b**s may be shared by many **a**s. To represent shared aggregation, a line joins classes **a** and **b**, and a hollow diamond terminates the line at **a**. A multiplicity (or cardinality) may be indicated at each end of the line, referring to which ever class the multiplicity is nearer. The multiplicity states how many (or what range of) objects of the given type may be aggregated with an object of the type given by the class at the other end of the aggregation line.

Shared aggregation may be implemented in C++ by having pointers as data members. The aggregated object may be modified once it has become part of the aggregating object, and that modification be propagated through to the instance of the class.

Composition aggregation

Composition aggregation is a refinement of shared aggregation which implies that an **a** exclusively contains a particular **b**. This may be implemented in C++ by having an actual data member rather than a pointer. When a data member is aggregated by value to an instance of a class, a local copy of the object is made, and destroyed when the instance of the class is destroyed.

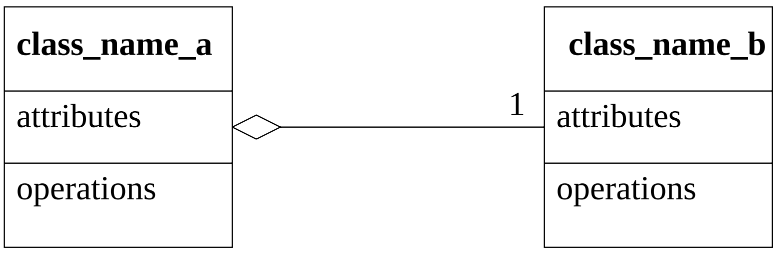

Figure A.3: Shared aggregation in UML. Here, **class_name_a** contains exactly one **class_name_b**, although this **class_name_b** may be shared by many instances of **class_name_a**. Composition aggregation is represented by a filled diamond.

A filled diamond may be used to express composition aggregation. The filled diamod may replace any hollow diamond with a multiplicity of one. An important distinction between shared and composition aggregation is that when aggregation is shared, instances of the aggregated classes (parts) may exist independently of the instances of the owner class (whole). For example individual simulation steps may or may not be reconstructed to form a track. When aggregation is by composition, the instances of the parts have lifespans which may not exceed that of the whole. Such behaviour is demonstrated by the *Functionality* object that forms part of a *G4VGraphicsSystem* object. The *Functionality* relates specifically to the *G4VGraphicsSystem*, and there would be little gained in making it available without a *G4VGraphicsSystem*.

Appendix B

Parameters for BaBar EMC shower data runs

The data collected from the Geant4 simulations performed in respect of the attempt to parameterize electromagnetic showers in CsI(Tl) were done so via the use of a model with the following properties.

Atomic number (Z)	7.3
Atomic mass (A)	14.610 $g/mole$
Density (ρ)	1.205×10^{-3} g/cm^3
Temperature (T)	293K
Pressure (P)	1 atmosphere
Magnetic field (B)	0 Tesla
Electric field (E)	0 V/m

Table B.1: Properties of the 'world volume' described to Geant4 for the simulation of electromagnetic shower data.

Atomic number (Z)	54.023
Atomic mass (A)	129.969 $g/mole$
Density (ρ)	4.51 g/cm^3
Temperature (T)	293K
Pressure (P)	1 atmosphere
Magnetic field (B)	0 Tesla
Electric field (E)	0 V/m

Table B.2: Properties of the CsI(Tl) crystal described to Geant4 for the simulation of electromagnetic shower data.

The CsI(Tl) crystal was a cylinder of radius and half-length 60cm. The cylinder was positioned with its axis parallel to the $z-axis$ of world coordinate space, and one planar-face perpendicular to the $z-axis$ at z=0.0cm. $e^{\pm}$, γ were fired parallel to the $z-axis$ into the crystal from the coordinates $(0.0, 0.0, 0.0)$ in world space.

Bibliography

[1] Katsuya Amako, Guiseppe Ballocchi and Peter Urban. *GEANT4 OO Toolkit for Particle Detector Simulation - User Requirements Document version 6.2.* CERN, European Laboratory for Particle Physics, May 1997.

[2] Katsuya Amako, Gunter Folger and Simone Gianni et al. *Geant4 User Guide For Application Developers.* Available on the WWW via http://wwwinfo.cern.ch/asd/geant/geant4_public/G4UsersDocuments, July 1998.

[3] Guy Barrand. The g4xt singleton class for handling xwindowing system events. Part of the Geant4 kernel.

[4] Frederic Delhoume (delhoume@ilog.fr). A gouraud shaded postscript representation of a triangle. Posted to comp.graphics.api.opengl by Mark Kilgard as part of Achieving Quality PostScript output for OpenGL.

[5] Anton Eliens. *Principles of Object-Oriented Software Development.* Addison-Wesley Publishing Company, 1994.

[6] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit.* John Wiley & Sons, Inc., 1998.

[7] C. Caso et al. *Particle Physics Booklet*, chapter 23, page 188. Springer Verlag, 1998.

[8] Satoshi Tanaka et al. *An Introduction to DAWN.* Fukui University, Japan, tanaka@i1nws1.fuis.fukui-u.ac.jp, 1997. Available via WWW at http://geant4.kek.jp/∼tanaka/DAWN/About_DAWN.html.

[9] Paul E. Kimball. *The X Toolkit cookbook.* Prentice Hall P T R., 1995.

[10] E. A. Maxwell. *Methods of Plain Projective Geometry Based on the Use of General Homogeneous Coordinates.* Cambridge University Press, Cambridge, England, 1946.

[11] Tom McReynolds and David Blythe, editors. *Programming with OpenGL : Advanced rendering.* SIGGRAPH, 1997.

[12] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall International Series in Computer Science, 1988.

[13] Mark Kilgard (mjk@fangio.asd.sgi.com). Achieving quality postscript output for opengl. Posted to comp.graphics.api.opengl newsgroup, April 1997. Available via WWW at http://reality.sgi.com/opengl/tips/Feedback.html.

[14] Brian Paul. *The Mesa 3D graphics library.* Available on the WWW via http://www.ssec.wisc.edu/∼brianp/Mesa.html, June 1998. An index of resources for Mesa.

[15] Mark Segal and Kurt Akeley. *The design of the OpenGL interface.* Silicon Graphics Computer Systems 2011 N. Shoreline Blvd., Mountain View, CA 94039, 1994.

[16] Bjarne Stroustrup. What is "object-oriented programming"? In G. Goos and J. Hartmanis, editors, *ECOOP '87 European Conference on Object-Oriented Programming*, volume 276 of *Lecture Notes in Computer Science*, pages 51–70. Springer-Verlag, 1987.

[17] Bjarne Stroustrup. *The C++ programming language second edition.* Addison-Wesley Publishing Company, Inc., 1991.