SANDIA REPORT
SAND98-8225 • UC–405
Unlimited Release
Printed January 1998

Quantify Uncertain Emergency Search Techniques (QUEST)—Theory and User's Guide

M. M. Johnson, M. E. Goldsby, T. D. Plantenga, T. L. Porter, T. H. West, W. B. Wilcox, and W. K. Hensley

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A07
Microfiche copy: A01

SAND98-8225
Unlimited Release
Printed January 1998

Distribution
Category UC-405

Quantify Uncertain Emergency Search Techniques (QUEST)—Theory and User's Guide

Michael M. Johnson*, Michael E. Goldsby, Todd D. Plantenga, Terry L. Porter,
Todd H. West, and William B. Wilcox
Systems Studies Department
Sandia National Laboratories
Livermore, California 94550

Walter K. Hensley
Nuclear Chemistry Section
Pacific Northwest National Laboratory
Richland, Washington 99352

ABSTRACT

As recent world events show, criminal and terrorist access to nuclear materials is a growing national concern. The national laboratories are taking the lead in developing technologies to counter these potential threats to our national security. Sandia National Laboratories, with support from Pacific Northwest National Laboratory and the Bechtel Nevada, Remote Sensing Laboratory, has developed QUEST (a model to Quantify Uncertain Emergency Search Techniques), to enhance the performance of organizations in the search for lost or stolen nuclear material. In addition, QUEST supports a wide range of other applications, such as environmental monitoring, nuclear facilities inspections, and searcher training.

QUEST simulates the search for nuclear materials and calculates detector response for various source types and locations. The probability of detecting a radioactive source during a search is a function of many different variables, including source type, search location and structure geometry (including shielding), search dynamics (path and speed), and detector type and size. Through calculation of dynamic detector response, QUEST makes possible quantitative comparisons of various sensor technologies and search patterns. The QUEST model can be used as a tool to examine the impact of new detector technologies, explore alternative search concepts, and provide interactive search/inspector training.

* email: mmjohns@ca.sandia.gov

ACKNOWLEDGEMENTS

The authors wish to thank T. Dahlstrom and R. Hansen of the Bechtel Nevada, Remote Sensing Laboratory for sharing their time and expertise throughout the development of QUEST. In addition to the paper's listed authors, our thanks go also to the many Sandians who contributed to QUEST's outcome, in particular C. A. Flores, K. J. Nisewaner, G. W. Richter, and T. J. Sa. We also acknowledge support by the Department of Energy through Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Peak-to-Compton ratios as well as multiple Compton regions were calculated using models developed by Hensley, et al., as published in "SYNTH, A Computer Code to Generate Synthetic Gamma Ray Spectra", Pacific Northwest National Laboratory, 1994.

Data for calculating the mass attenuation coefficients of absorber materials were taken from the "Radiological Health Handbook", published by U.S. Department of Health Education and Welfare.

Efficiency curves for Germanium detectors are calculated using algorithms developed by Gunnink and Prindle, as published in "Nonconventional Methods for Accurately Calibrating Germanium Detectors", UCRL-JC-105283, Lawrence Livermore National Laboratory, 1991.

Spectral peak shapes were calculated using models developed by Gunnink and Niday, and published in "Computerized Quantitative Analysis by Gamma Ray Spectrometry, Volume 1, Description of the GAMMANAL Program", UCRL-51061, Lawrence Livermore National Laboratory, 1972.

The shape of the Compton continuum was generated using the algorithms of Kopecky, Ratynski, and Warming, as published in "Curves for the Response of a Ge(Li) Detector to Gamma Rays up to 11 MeV" NIM, 50:333.

Master Library used with permission granted to the author by
Gerhard Erdtmann, Werner Soyka
Forschungszentrum Juelich GmbH
Zentralabteilung fuer Chemische Analysen
Postfach 1913 - D-5170 Juelich
Verlag Chemie
Postfach 1260/1280 - D-6940 Weinheim.

Other gamma-ray libraries and nuclear data used with permission of their respective authors.

QUEST utilizes the commercial graphics library World ToolKit™. World ToolKit is a registered Trademark of Sense8 Corporation, Mill Valley, CA 415/331-6318.

CONTENTS

1 Introduction 7
2 User's Guide 9
2.1 Scenario Definition Mode 10
2.1.1 Structures 11
2.1.2 Sources 16
2.1.3 Path Definition 20
2.1.4 Detectors 21
2.1.5 Background 29
2.2 Simulation Mode 30
2.2.1 Simulation Control 30
2.2.2 Performance Tuning 31
2.3 Analysis Mode 34
3 Theory 36
3.1 The Physics of Gamma Ray Scattering 36
3.1.1 Attenuation Effects 39
3.2 Transport Physics 42
3.2.1 Algorithm Overview 42
3.2.2 Software Design Overview 43
3.2.3 Accounting for Source and Detector Motion 44
3.2.4 Calculating Losses Between Source and Detector 45
3.2.5 Modeling Detector Physics 49
3.2.6 Modeling Detector Signal Processing 55
3.3 Material Database 57
3.3.1 Total Thickness 57
3.3.2 Density 58
3.3.3 Elemental Composition 58
3.4 Background Radiation 60
3.5 Component Database 63
3.5.1 Binary Space Partition Tree 64
3.5.2 Structural Component Grouping 80
3.6 Interprocess Communication 82
3.7 Graphics Rendering Engine 86
3.7.1 Structures 87
3.7.2 Path Specification 90
3.7.3 Structural Component Picking 92
3.7.4 Lighting 92
3.8 Graphical User Interface 94
4 A Comparison of Measured and Synthetic Response Functions 97
4.1 Comparison of Spectra 97
4.2 Comparison of Algorithm Responses 100
5 Summary 103

References 104
Bibliography 106

Appendix A: Software Installation 110
Appendix B: Structure Creation 112
Appendix C: Raw Building Materials 117
Appendix D: Component Materials 122
Appendix E: Radionuclide Library 128

Intentionally Left Blank

1 INTRODUCTION

There are many challenges associated with the search for nuclear material (Figure 1.1). The probability of detecting a radiological source is a function of many different variables. Moreover, probabilities of detection (PD) are unknown quantities; while the probability of detection is assumed to be high based on known detection ranges for an unshielded source, no quantitative estimates are available. Search requirements are often based on experience and intuition. Typical questions for which a quantitative analysis is necessary include:

- For a given source, structure, search pattern, and detector, what is the probability of detection? If no detector signal is received and PD is 0.95, the searcher could continue somewhere else. If no detector signal is received and PD is 0.45, perhaps the structure should be searched more thoroughly, or new detector technologies pursued.

- For a given source, structure, and detector, what is the optimum way to search? If searching both sides of a hallway increases PD from 0.45 to 0.9, then it would be worthwhile. If, on the other hand, searching both sides of the hallway increases PD from 0.95 to 0.97, then it is probably not worth expending the extra search time.

- What are the payoffs from new technologies and designs? If doubling the detector area increases PD from 0.45 to 0.9, then it is worthwhile investigating lighter detectors. If doubling detector area increases PD from 0.45 to 0.5, then it is not worthwhile to investigate lighter detectors.

QUEST provides a tool to answer these questions and others like them by simulating all aspects of the search process. QUEST provides the ability to call up computer models of many different building types in which a nuclear device may be concealed. These structures can be developed using any one of many different Computer Aided Design (CAD) packages. Once a structure has been developed, the user can import the building design into the QUEST search simulation. The primary QUEST window displays the first-person, three-dimensional (3D) point-of-view (POV) of the searcher (see Figure 2.7). A second window in the upper corner of the display shows the two-dimensional floor plan. The user then specifies the characteristics of the nuclear material source (for example, a nuclear weapon or radiation dispersal device) within the floor plan and selects a detector model (such as a hand-held radiation sensor). Once the simulation begins, the user can move throughout the simulated environment, analyzing the calculated detector response.

Figure 1.1: The search for nuclear material.

QUEST is also capable of synthesizing the results of typical gamma-ray spectroscopy experiments. Specifically, QUEST allows a user to specify physical characteristics of a gamma-ray source, the quantity of the nuclides producing radiation, the structure and type of absorbers, the size and composition of the detector (Ge or NaI), the electronic setup used to gather the data, and background terms associated with the simulation. In the process of specifying the parameters needed to synthesize a spectrum, several interesting intermediate results are produced, including a photopeak transmission as a function of energy, a detector efficiency curve, and a weighted list of gamma and x-rays produced from a set of nuclides. All of these intermediate results are made available to the user.

QUEST is a multithreaded application that runs on UNIX workstations and PC compatible computers, including multiprocessor servers and portable computer systems. QUEST contains robust support for germanium detectors, and support for specific sodium iodide detectors. Spectra generated with QUEST have been compared to spectra obtained from National Institute for Standards and Technology (NIST) certified standards with very favorable results. A discussion of the use of QUEST, together with the theory behind its design and implementation, is presented.

2 USER'S GUIDE

The QUEST application is divided into three distinct modes of operation: Scenario Definition, Simulation, and Analysis. Scenario Definition mode allows the user to specify all the components necessary to perform a search simulation (e.g. structure specification, source and detectors, etc.). Simulation mode executes the actual search simulation utilizing the user-specified scenario. And finally, Analysis mode allows the user to step back through data history files collected during a previous simulation run in order to perform more detailed analysis. The following three subsections describe the user interface and functionality of these three modes of operation.

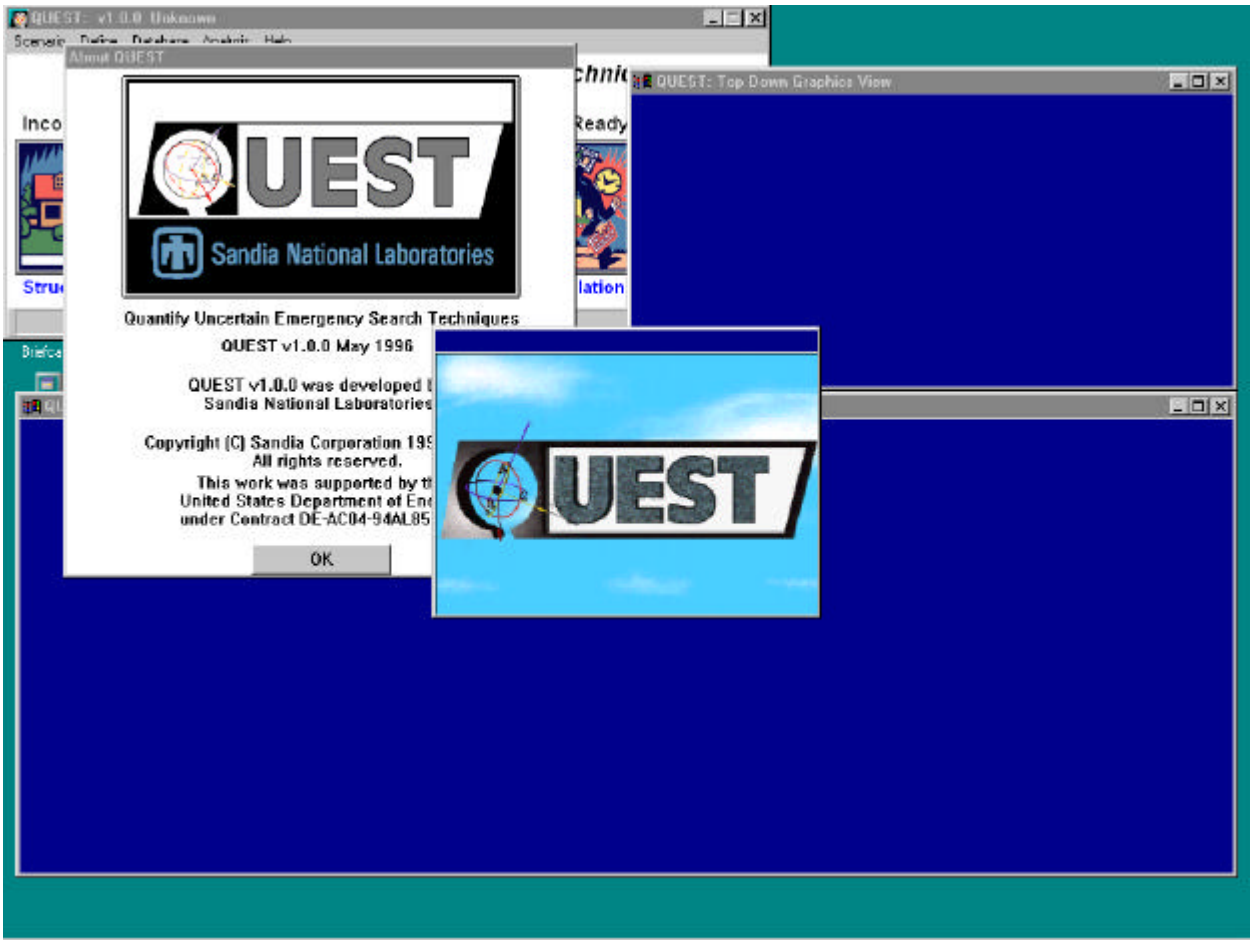


Figure 2.1: QUEST application start-up.

When QUEST is first started a number of windows appear (Figure 2.1). These include the QUEST copyright window, the primary user interface window encompassing the six control buttons, two three-dimensional display windows that are initially blue, and the QUEST sign-on animation window. Once the opening animation sequence is complete, and the QUEST copyright window has been closed, the user may begin interacting with the user interface controls.

Figure 2.2: QUEST primary window.

The primary Graphical User Interface (GUI) is a single windowpane comprising several drop down menus, and a series of buttons (Figure 2.2). The drop down menus control storage and retrieval of Scenario definitions (Scenario), specification of a radiological database (Database), selection of output from a previous simulation run for analysis (Analysis), and selection of application information screens (Help). The series of buttons provide access to the three modes of operation. Scenario Definition is comprised of Structure(s), Source(s), Detector(s) and Background specification. When QUEST is first started, each of these subcomponents is not yet complete, and so each bears the label "Incomplete". In turn, as each of these Scenario subcomponents is completed, the labels change to "Complete". In a like fashion, the labels above the Simulation and Analysis mode buttons will change from "Not Ready" to "Ready" as each of the previous subsections are completed. Simulation mode becomes "Ready" when a Scenario has been completely defined (i.e. all four components, Structure(s), Source(s), Detector(s), and Background have been defined). Analysis mode becomes "Ready" when a simulation run has been completed, and there is simulation data to be analyzed.

Throughout QUEST, object specifications and generated data are stored to computer disk files according to a predetermined directory structure. The imposed directory structure ensures that all object data files are interpreted appropriately at run-time, eliminating a great deal of run-time object checking from the application. While this restriction will be eliminated in future releases of the application, its presence in the initial version of QUEST serves to simplify component object organization and access. Hence, all QUEST objects are saved and retrieved from predetermined subdirectories, the locations for which are always fixed relative to the parent directory of the QUEST application. For example, when operating within the QUEST environment, the user is always assured of finding all defined source specifications in the source subdirectory. For a more detailed treatment of the QUEST directory structure see Appendix A: Software Installation.

2.1 SCENARIO DEFINITION MODE

A Scenario encompasses all of the parameters required to execute a search simulation, and is comprised of the following:

- Structure Layout
  - Structure Definition
  - Material Database
  - Component Grouping & Material Assignment

Source Definition
    Physical Composition
    Radioisotope Specification
Detector Definition
    Detector Type & Material Specification
    Electronics
    Algorithms
Background Definition

Under this organization, like parameters are grouped in a natural order. For instance, all the parameters related to the detector specification are together. Each of the four primary subsections, Structure Layout, Source Definition, Detector Definitions, and Background, is accessed through its corresponding button in the QUEST Primary window (Figure 2.2). This grouping makes it convenient for QUEST to display information that is traditionally computed, such as a photon transmission curve or a detector efficiency curve.

The title bar of the Primary window displays the version number of QUEST being executed, as well as the name of the current scenario. If a scenario has not yet been defined, the scenario name is listed as "Unknown". The *Scenario* pull-down menu provides for the creation of a new scenario, "New", or file manipulation of completed scenarios through the "Load", "Save", and "Save As" choices. In addition, the *Scenario* pull-down menu contains the "Exit" menu choice for quitting the QUEST application all together. The *Database* pull-down menu provides access to the database manipulation routines. QUEST v1 does not support direct manipulation of the gamma-ray or nuclides databases. However, the user may modify the database outside of the application (see Appendix E: Radionuclide Library). The *Analysis* pull-down menu allows the user to jump immediately to the Analysis Mode in order to analyze results stored from a previous simulation run (see Section 2.3: Analysis Mode). Finally, the *Help* pull-down menu provides user access to application information screens including the *About QUEST*, *QUEST Credits*, and *QUEST Disclaimer*.

The following five subsections discuss the development of a QUEST scenario through use of the first four buttons on the primary QUEST window. Once defined or loaded, scenarios can be directly run through the *Simulation* panel button, and/or saved to disk file for later execution.

2.1.1 STRUCTURES

With a click of the *Structure(s)* button, the *Structure Configuration* window appears (Figure 2.3). Through the *Structure Configuration* window, the user may load and convert structure files stored on disk in either three-dimensional DXF (AutoDesk 1990) or VRML v1 (Bell et al. 1995) compliant QUEST Structure File (QSF) formats.

A library of structure files is included with QUEST. Appendix A (Software Installation) details these structures, their file names, and approximate display complexity specified as the number of polygons used to render them. The provided structure library includes simple structures for use in quick "what-if" comparison studies, as well as enhanced models demonstrating structure detail such as door casings. All structure library models are provided in both original 3D DXF and QSF file formats. The original 3D DXF structure library models can be used with a commercial CAD package as starting-points in the development of additional models. For a detailed treatment of structure file creation using a third-party CAD application refer to Appendix B: Structure Creation Guide.

The three buttons located on the *Structure Configuration* window provide access to the three subcomponents that must be defined for each structure. As with the *QUEST Primary* window, a label above each button indicates the status of the definition for that subsection. Once a subsection has been defined, the label changes from "Incomplete" to "Complete". Once all three subsections are complete, structure definition is complete.

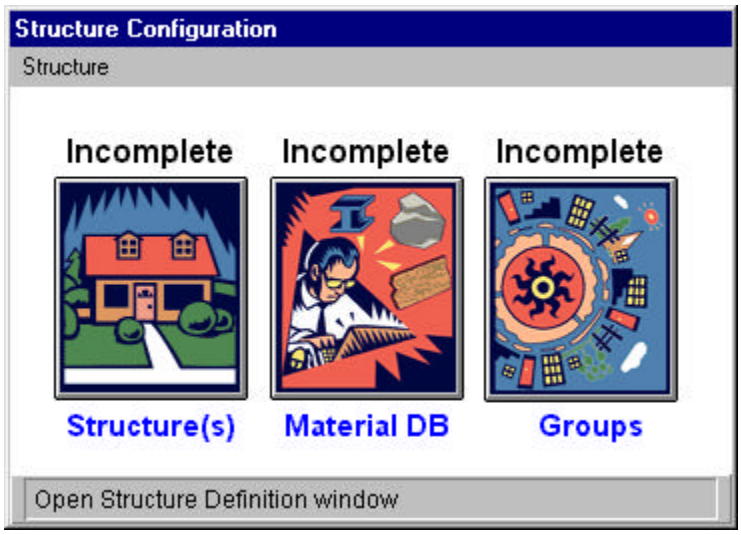


Figure 2.3: Structure configuration window.

Structure Definition

Selection of the *Structure(s)* button from the *Structure Configuration* window brings up the *Structure Definition* window (Figure 2.4). The three pull down menus of the *Structure Definition* window provide access to unit measure specification and CAD file conversion routines. The *Window* pull-down menu provides the option of closing the *Structure Definition* window. *Units* provides the unit measure choices for the application. All QUEST v1 structure files are manipulated in units of feet. And finally, *Convert* accesses the 3D DXF CAD file conversion routines.

To convert a 3D DXF CAD file, select the *Convert* pull-down menu from the *Structure Definition* window. A list of available files is presented. Once selected, the 3D DXF CAD file is converted into QUEST's internal QSF structure format and displayed for review in the 3D graphics window. In order to reduce the complexity in merging multiple CAD file specifications, 3D DXF CAD files must be converted into QSF files one-at-a-time.

Structure definition involves the grouping of structure components into a hierarchy. To support this, the *Structure Definition* windowpane is divided into two halves. The left-hand side of the screen provides user access to structure file manipulation routines, including translations and rotations of substructure components. The right-hand side of the screen displays the current hierarchy of substructure components associated with the current structure definition session. The *Structure Name* pull-down list-box provides access to all available QSF structure file components. Once selected under *Structure Name*, a structure's description is displayed for review in the *Structure Description* scroll window. The substructure component can then be placed into the current scenario by highlighting the desired position in the structure hierarchy display on the right-hand side of the screen, and pressing *Insert New Structure*. Likewise, a structure subcomponent can be removed from the hierarchy by highlighting it and pressing *Remove Structure*.

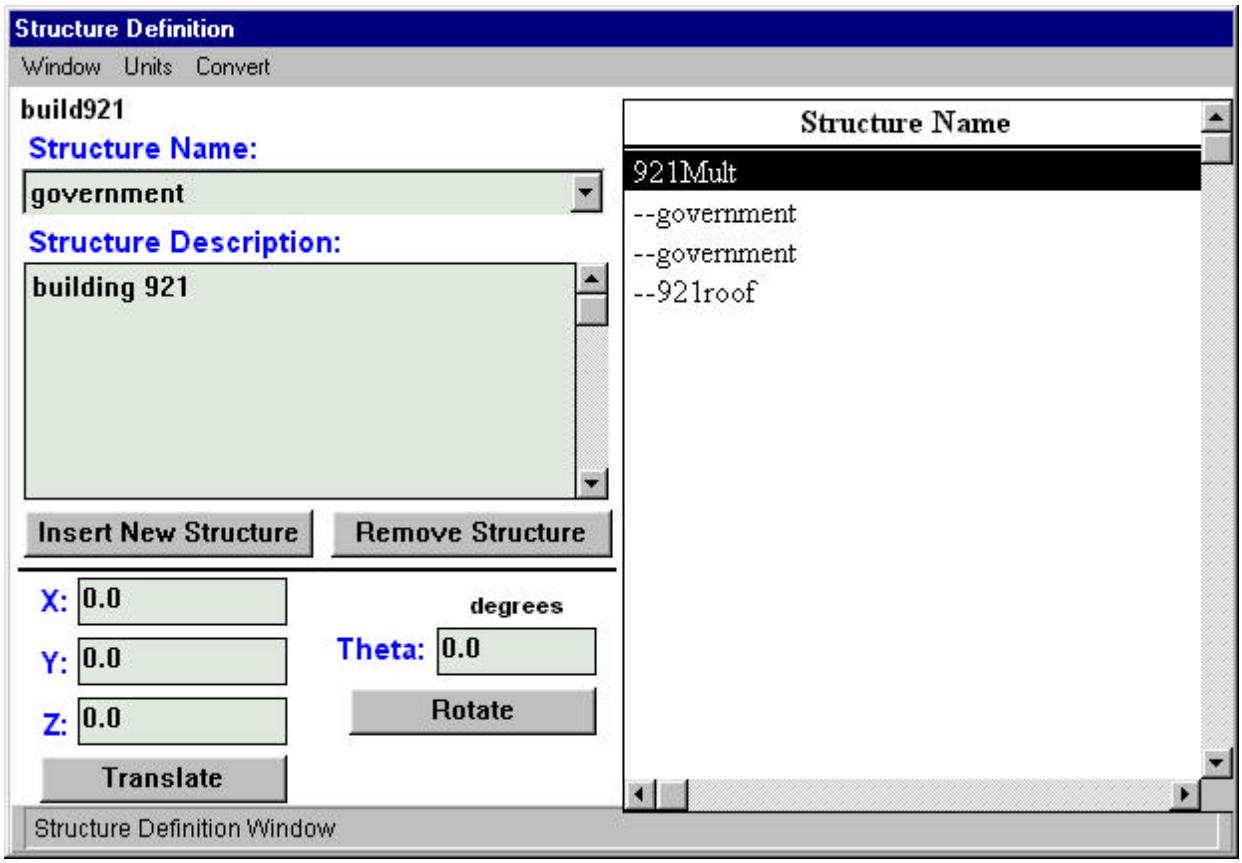


Figure 2.4: Structure definition window.

At any time during the creation of a structure hierarchy, individual structures can be manipulated in the virtual environment. QUEST v1 supports translations and rotations. To translate a structure subcomponent, select the structure component in the right-hand window, enter the translations values (*X*, *Y*, and *Z*), and then press *Translate*. Likewise, to rotate a structure subcomponent, select the subcomponent, enter a rotation value in degrees (Theta), and press *Rotate*.

Material Database

Selection of the *Material DB* button from the *Structure Configuration* window brings up the *Material Database* window (Figure 2.5). The three pull-down menus of the *Material Database* window provide access to material manipulation routines. Specifically, the *Database* pull-down menu provides for the creation of a new material database, "New", or file manipulation of completed material databases through the "Load", "Save", and "Save As" choices. In addition, the *Database* pull-down menu contains the "Close" menu choice for exiting the *Material Database* window. The *Material* pull-down menu accesses the individual material *Insert* and *Remove* operations, supporting the manipulation of material instances within the current material database. And finally, the *Options* pull-down menu supports the toggle of the material transmission chart between a *Linear Plot* and *Log Plot*.

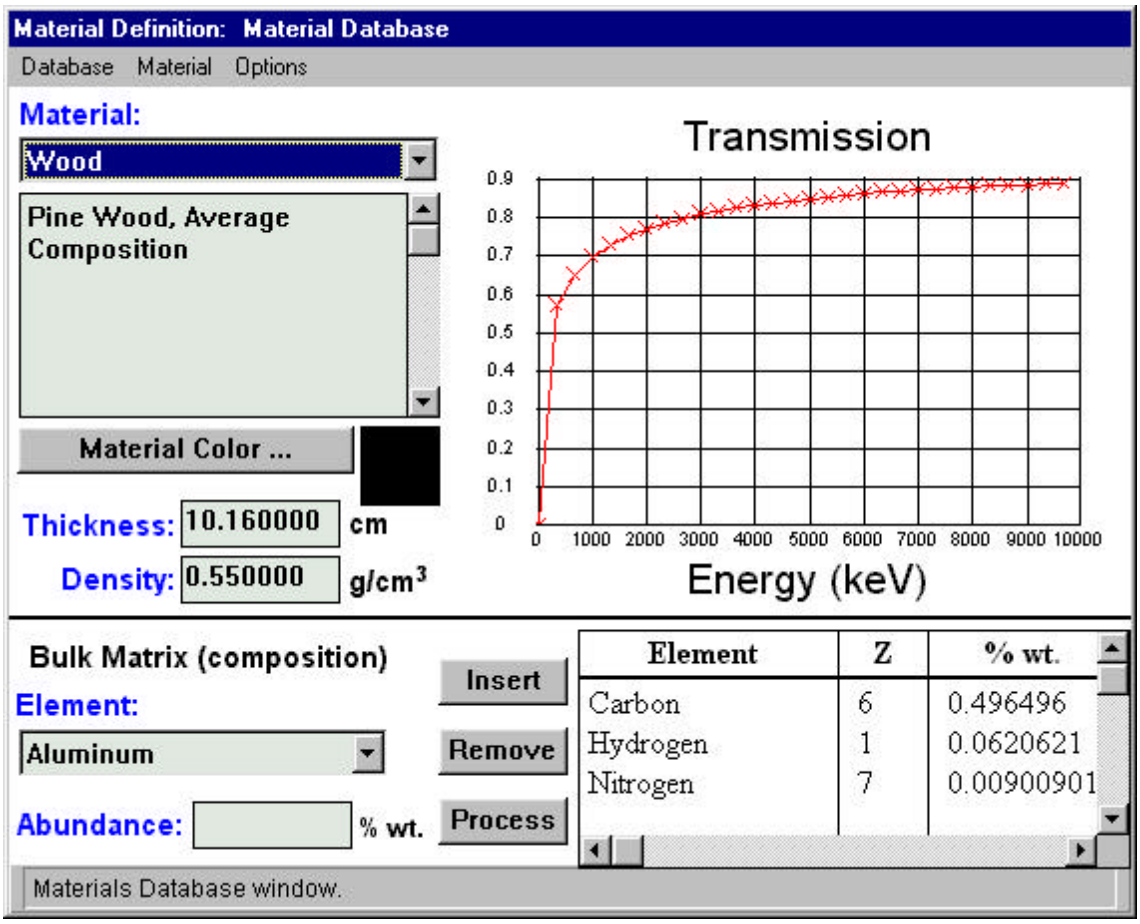


Figure 2.5: Material database window.

A material is specified under the *Material Database* window through the combination of five attributes: name, color, thickness, density, and bulk matrix specification. The name specified for the material must be unique for that material database. In addition, the assigned material color, accessed through the *Material Color* button, is displayed for objects during group picking and material assignment mode (see Component Grouping Definition below).

To insert a new material, select *Insert* from the *Material* pull-down menu of the *Material Database* window. Once a new material is inserted, modify its attributes through selection of its color (*Material Color*), thickness (cm), and density (g/cm^3). In addition, the bulk matrix composition for the material must be specified. This is given as an admixture of elements and their relative abundance by weight within the material. Once entered, the mixture is normalized, and the coefficient of absorption transmission chart is updated through selection of the *Process* button. For a more detailed treatment of material specification, see Section 3.3 Material Database.

Component Grouping Definition

Selection of the *Groups* button from the *Structure Configuration* window brings up the *Component Grouping Definition* window. Under *Component Grouping Definition* structural objects within the virtual environment may be grouped by physical or material attributes. The *Window* pull-down menu provides the option of closing the *Component Grouping Definition* window.

Component groups are the mechanism by which QUEST associates physical attributes with the structure components that make up QUEST's virtual environment. This mechanism was necessitated by the lack of standardized material assignment standards in CAD drawing packages. A group is added or removed from the

collection of named groups through the selection of the *Insert New Group* and *Remove Group* buttons respectively. Since the groups associated with a structure comprise a hierarchy of attributes, the location of insertion within that hierarchy must be selected prior to group insertion.

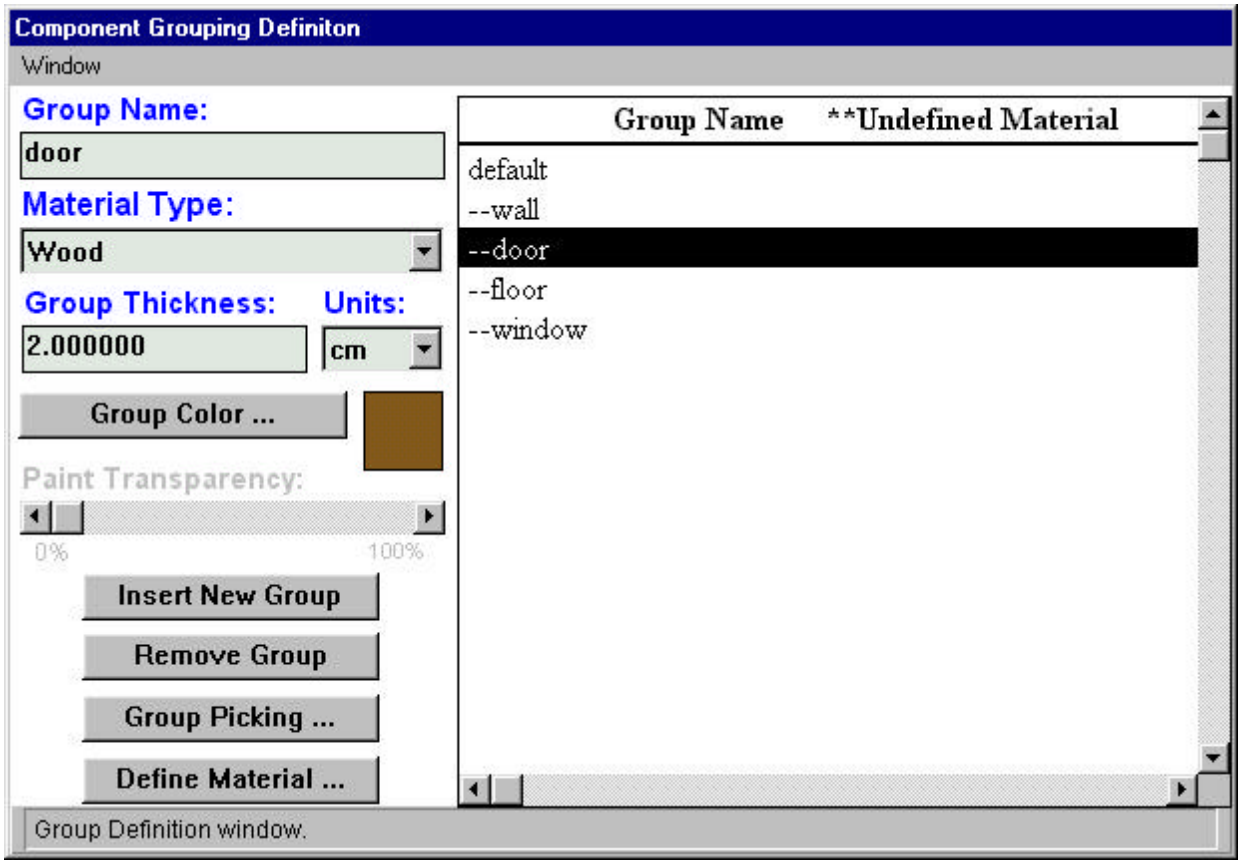


Figure 2.6: Component grouping definition window.

Once inserted, a group can be assigned a default material, group thickness, and color. Future versions of QUEST will support the *Paint Transparency* attribute, simulating more realistically transparent objects such as glass.

The hierarchy of groups within the *Component Grouping Definition* is significant as it defines the hierarchy of inherited attributes. In the example presented in Figure 2.6, if group "door" did not specify a material type, the material type would be inherited from its parent group, in this case "default". Once defined, the group hierarchy provides a quick and simple mechanism for manipulating physical attributes of the structure components within the virtual environment. Take, once again, the example presented in Figure 2.6, and consider a structure where all doors of a structure have been appropriately assigned the group name "door". Then it is a simple task to perform a simulation study of a given structure in which all doors are first of material type wood, and then changed to material of type steel. This can be accomplished through a single material attribute change of group "door".

Groups defined in the *Component Grouping Definition* window are assigned structure components, actual polygons within the virtual environment, through selection of *Group Picking*. *Group Picking*, as depicted in Figure 2.7, displays a specialized view of the structure environment by highlighting each polygon surface. The user may then select individual surfaces by placing the cursor over the polygon in the 3D graphics window and pressing the spacebar on the keyboard. Once selected, a structural component, or polygon, becomes a member of the assigned group and changes color to reflect its new association.

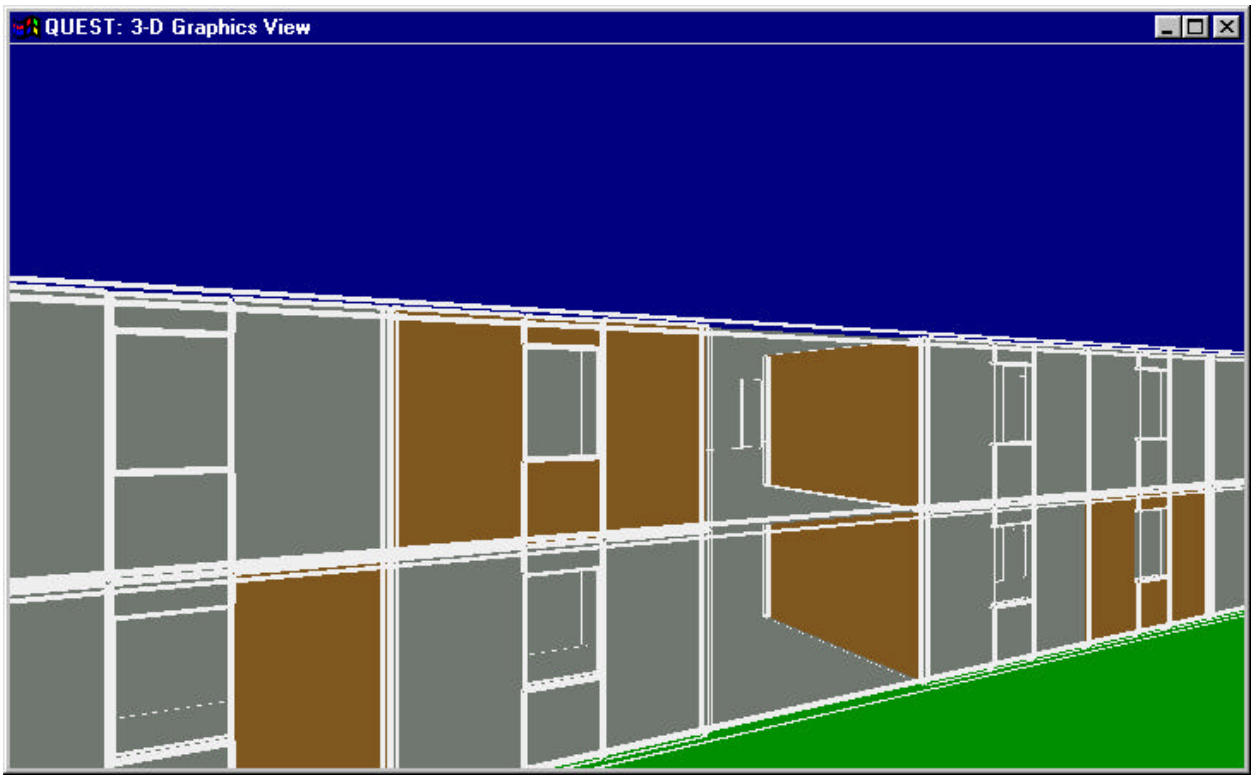


Figure 2.7: An example of structure component picking.

For convenience, the *Define Material* button provides access to the *Material Database* window for defining additional materials while performing component grouping.

2.1.2 SOURCES

With the selection of the *Source(s)* button from the *QUEST Primary* window, the *Source Path Assignment* window becomes visible (Figure 2.8). Within this window, the user matches Sources to Paths, with the matching displayed in a table on the right-hand side of the window. To match a previously defined Source and Path, and insert this association into the Scenario, the desired Source and Path must be selected from the Source and Path selection pull-down windows. Located in the middle of the Source Path Assignment window, the Source and Path selection pull-down menus list all previously defined and available Sources and Paths for inclusion within the current Scenario. Listed below each of these selection windows is a description given to the specified object, either source or path, assigned by the user at the time of definition. A previously entered source-path assignment can be removed from the table on the right by highlighting the assignment and selecting *Remove Source/Path*.

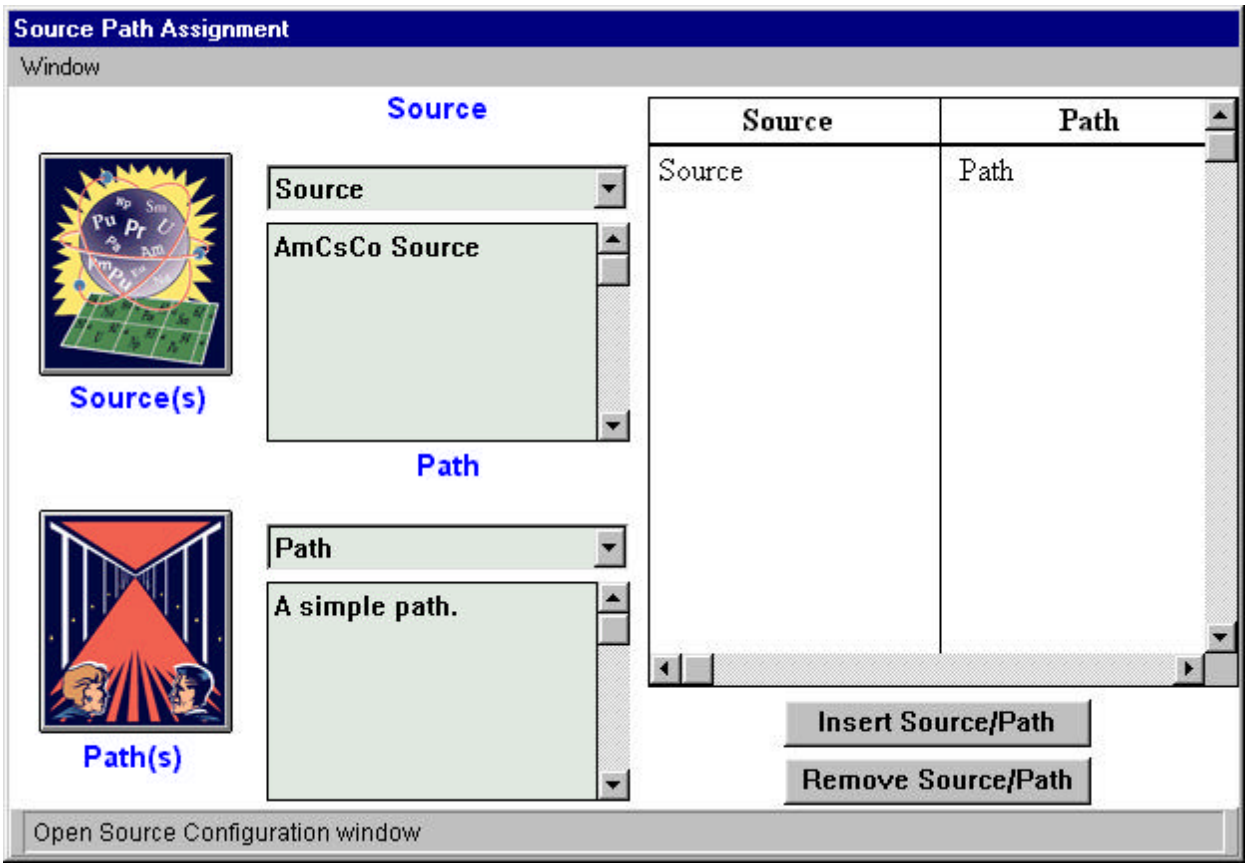


Figure 2.8: Source path assignment window.

The *Source Path Assignment* window includes two additional buttons on the left-hand side of the window for defining or editing Source and Path definitions. To edit the definition of an existing Source or Path, the user need only select the desired Source or Path, and then push the appropriate button. To create a new Source or Path, the user must push the appropriate button and then select "New" from the pull-down menu that appears under the Source or Path Definition window.

Selecting the *Source(s)* button on the *Source Path Assignment* window brings up the *Source Configuration* window (Figure 2.9). The *Source Configuration* window provides access to the two subcomponents that make up every source. Displayed in the title-bar of this window is the name of the source currently being manipulated. The *Window* pull-down menu provides the option of closing the *Source Configuration* window. The specification of a source is divided into two parts: physical sample properties and radionuclide specification.

Source Physical Composition

Selecting the *Sample* button of the *Source Configuration* window brings up the *Source Sample* window. Currently, QUEST v1 only supports point sources. However, work is underway to include disk shapes, where the bulk composition and thickness of the source determine the self-absorption. For finite sources, the material specification will include an admixture of elements and a user selected density. The *Window* pull-down menu provides the option of closing the *Source Sample* window.

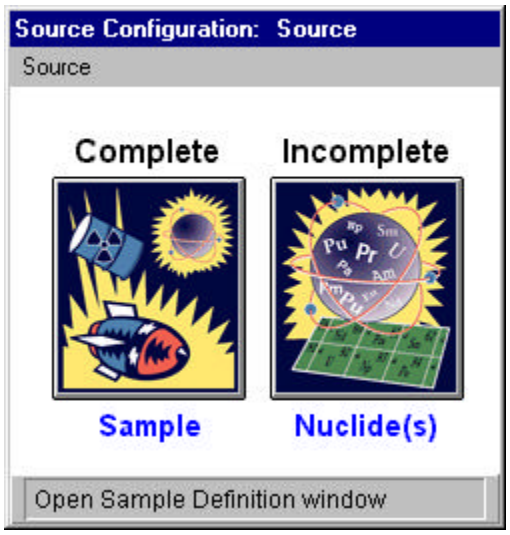


Figure 2.9: Source configuration window.

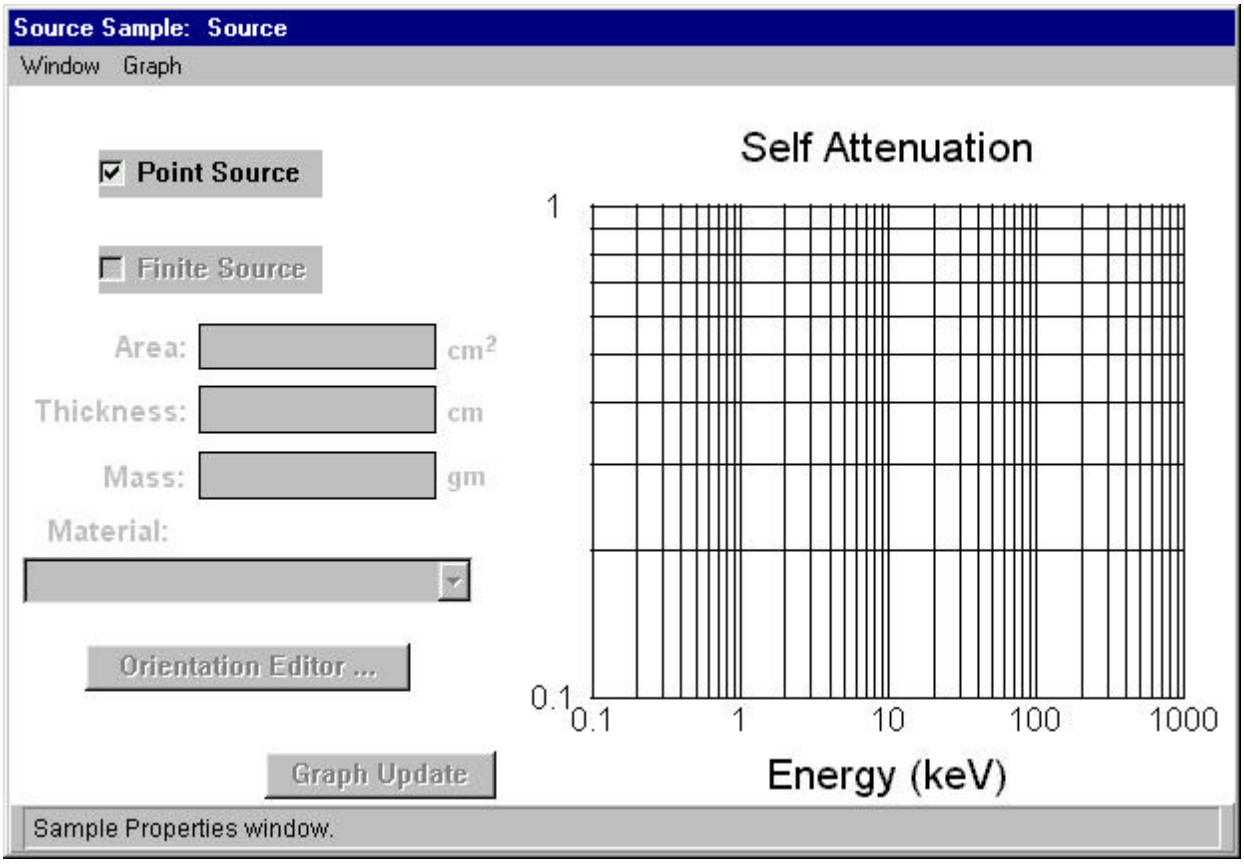


Figure 2.10: Source sample window.

Source Radioisotope Specification

Selecting the *Nuclide(s)* button from the *Source Configuration* window brings up the *Source Definition* window (Figure 2.11). This second portion of the source specification selects the radioisotopes contained in the sample. The pull-down menus of the *Source Definition* window provide access to database selection and sample radiosity specification. The *Window* pull-down menu provides the option of closing the *Source Definition* window. The *Database* pull-down menu provides for the selection of a radionuclide database—only the Erdtmann-Soyka (1979) database is provided with QUEST v1. And finally, the *Radiosity* pull-down menu accesses the *Source Radiation* definition window.

To support the specification of arbitrary radioactive sources, the Erdtmann-Soyka gamma-ray library has been incorporated into a relational database as have certain other data (e.g. parent-daughter branching ratios for selected nuclei, and stable isotopes along with their thermal and resonance neutron-capture cross sections) that were not included in the original compilation.

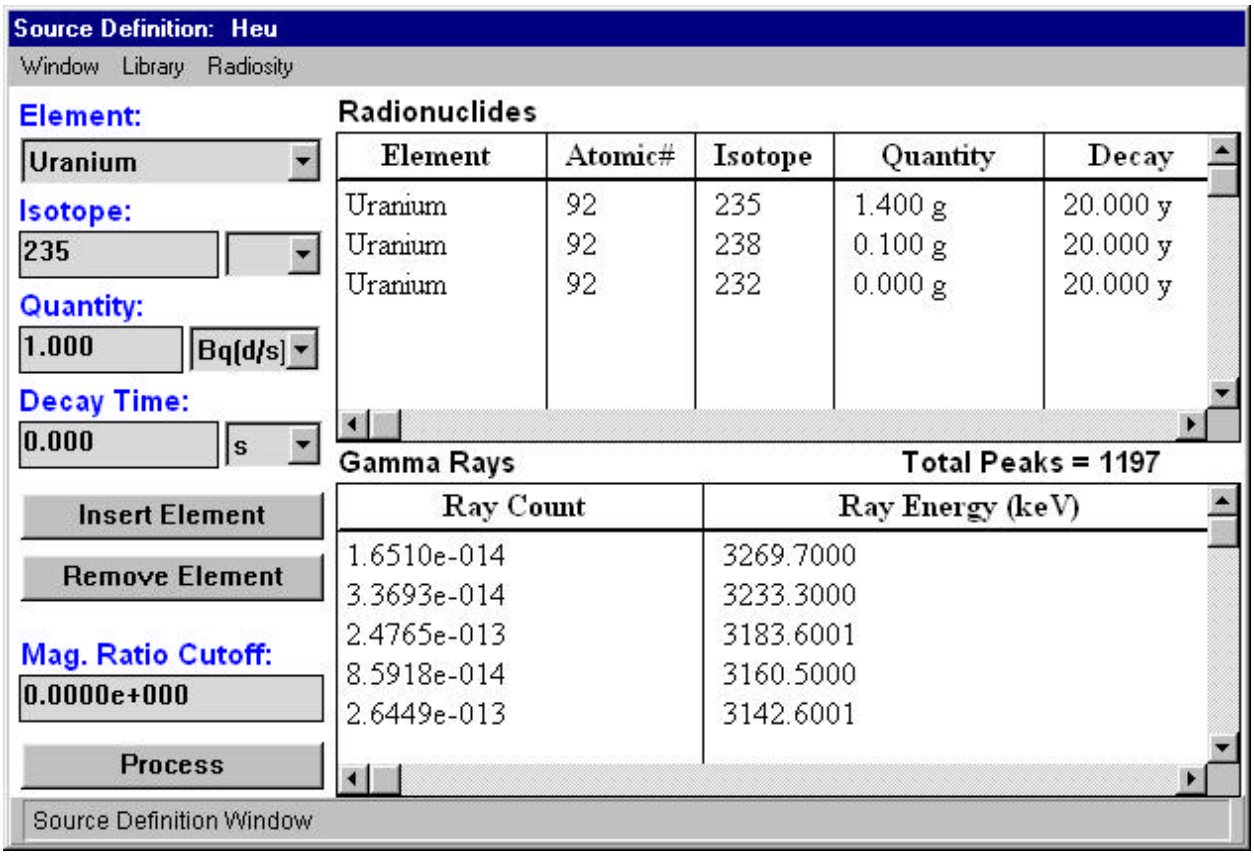


Figure 2.11: Source definition window.

In entering the radioisotope specification for a source, the user specifies the element (Z), atomic mass (A), and state (ground or metastable). As isotopes are added to a list, the quantity of the isotope is specified in one of several units. The completed isotope list may be subjected to decay, with daughter products automatically added to the list based on the specified decay time. Bateman equations are used to calculate the quantities of daughter products, and thus complex decay chains are modeled (Friedlander et al. 1981). With the selection of the *Process* button, QUEST searches the nuclide and gamma-ray databases and produces a sorted list of the isotopes contributing to the spectrum and the number of contributing gamma-rays.

The *Magnitude Ratio Cutoff* field specifies a value for limiting the number of gamma-rays that make up a source specification. This feature is provided to enhance real-time simulation performance. The magnitude ratio is used

to eliminate ray counts that make a minor contribution. Energies with ray count less than the ray energy with maximum ray count, divided by the magnitude ratio, are eliminated from the specification.

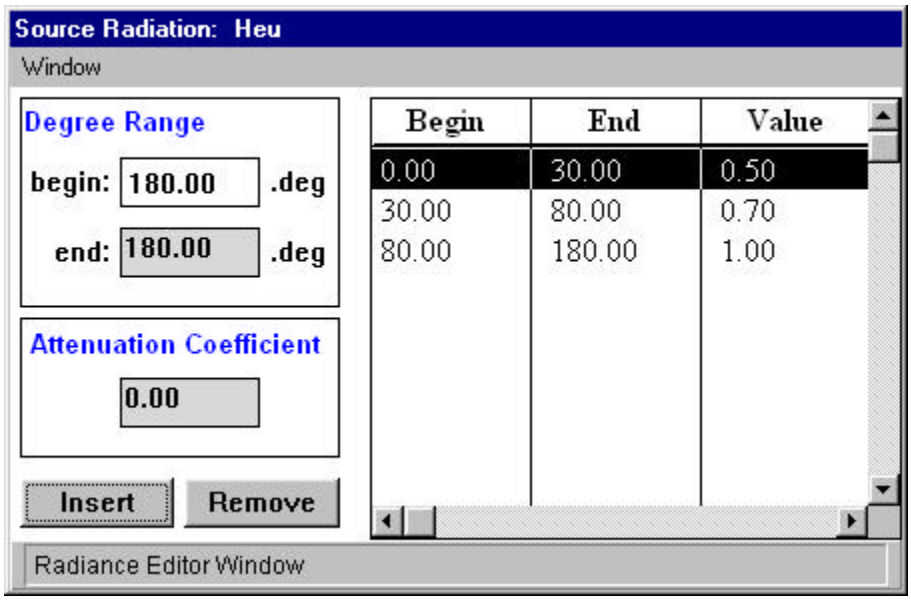


Figure 2.12: Source radiosity definition window.

The *Radiosity* pull-down menu of the *Source Definition* window accesses the *Source Radiation* definition window (Figure 13). An anisotropic source can be specified as a radially symmetric collection of attenuation coefficients. The default setting, 0—180 degrees with an attenuation coefficient of 1.0 specifies an isotropic source.

2.1.3 PATH DEFINITION

Selecting the *Path(s)* button from either the *Source Path Assignment* or *Detector Path Assignment* windows brings up the *Path Definition* window (Figure 2.13). Through the *Path Definition* window, a path can be immersively defined through the virtual environment. To define a path, select the *Record* button, and move through the environment using the 3D graphics window and the mouse. Once the path definition is complete, the *Stop* button stops path recording. Once a path has been recorded, it can be played back through selection of the *Playback* mode (Figure 2.14). Under *Playback* mode, controls are provided to move through the virtual environment on the previously defined path.

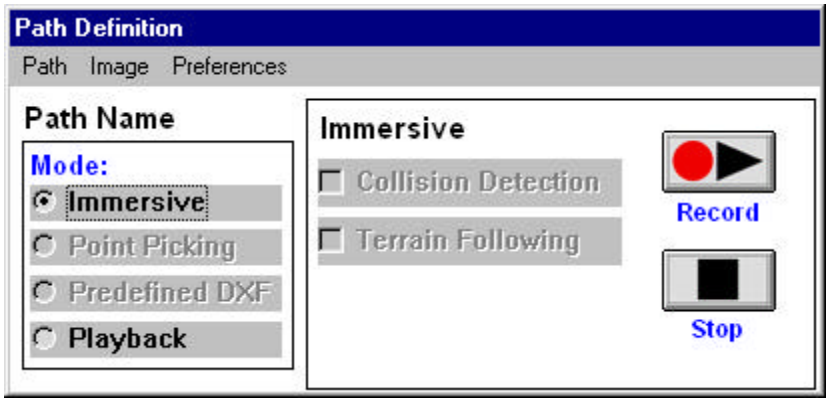


Figure 2.13: Path definition window—record.

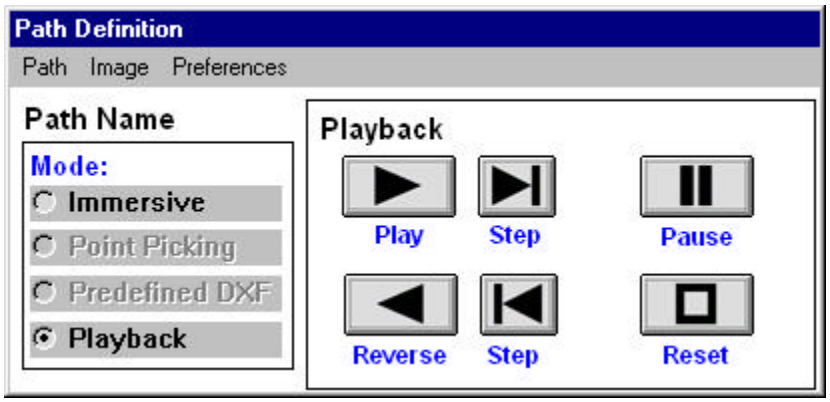


Figure 2.14: Path definition window—playback.

2.1.4 DETECTORS

The detector definition portion of QUEST presently supports a combination of a general algorithm for coaxial germanium diodes and efficiency curves for specific sodium iodide detectors. Though only five sodium iodide sizes are currently supported, work is underway to develop an algorithmic model based on results of Monte-Carlo derived response functions for sodium iodide crystals ranging in size from 1 in. x 0.5 in. up to 10 in. x 10 in. A complete range of coaxial germanium detector sizes is supported, from about 20% to 100% relative efficiency (compared to a 3 in. x 3 in. sodium iodide crystal at 1332 keV). Specifically, the germanium intrinsic detector efficiency is computed from fundamental detector parameters such as diameter, length, and relative efficiency using an algorithm developed by Gunnink and Prindle (1992). An absolute efficiency is then obtained by applying geometry factors to the computed intrinsic efficiency. Gunnink and Niday (1972) developed the detector geometry model as part of the GAMMANAL code.

Selecting the *Detector(s)* button from the *Primary QUEST* window displays the *Detector Path Assignment* window (Figure 2.15). This window is similar to the *Source Path Assignment* window, and allows for the association of an arbitrary number of detectors and paths.

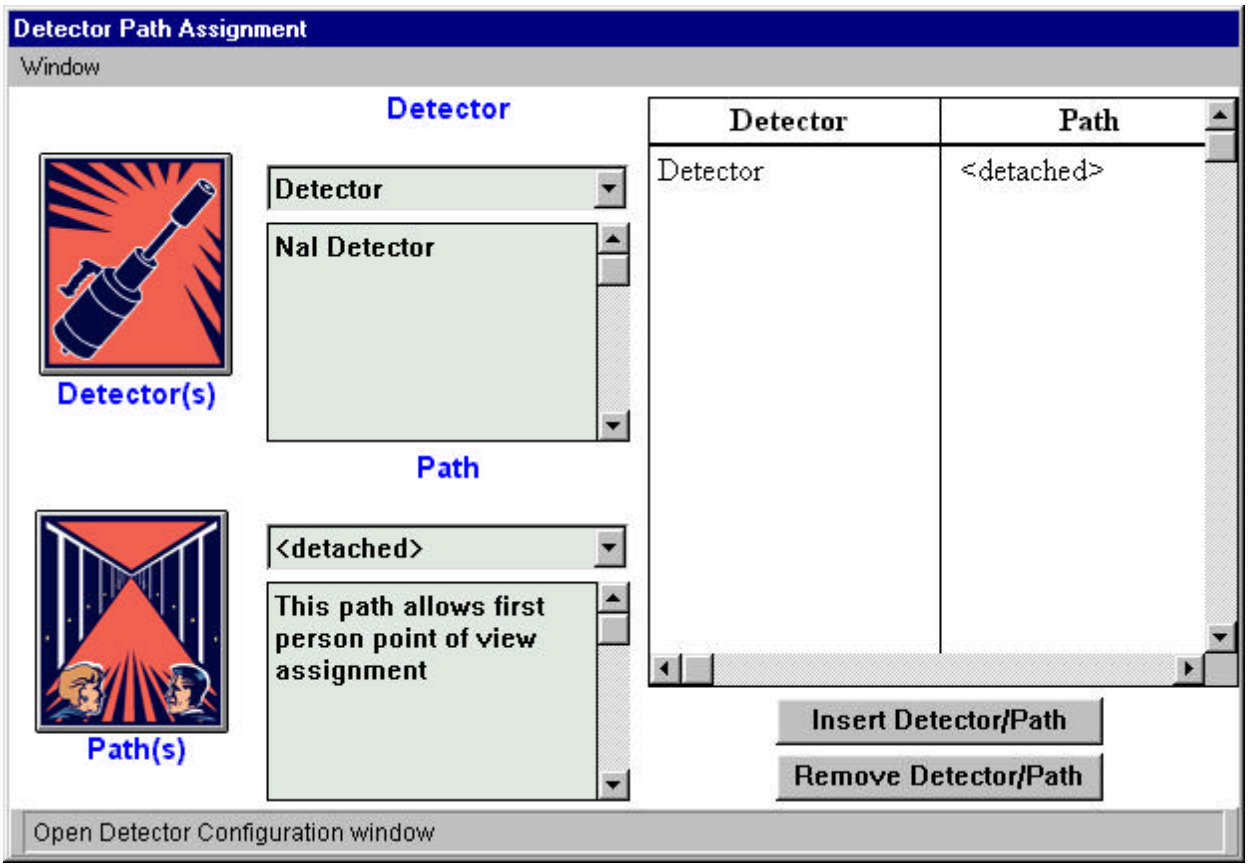


Figure 2.15: Detector path assignment window.

The *Path(s)* button of the *Detector Path Assignment* window brings up the *Path Definition* window (see Section 2.1.3). QUEST makes no distinction between source and detector paths. In addition, any number of objects, both sources and detectors, can be associated with a single path.

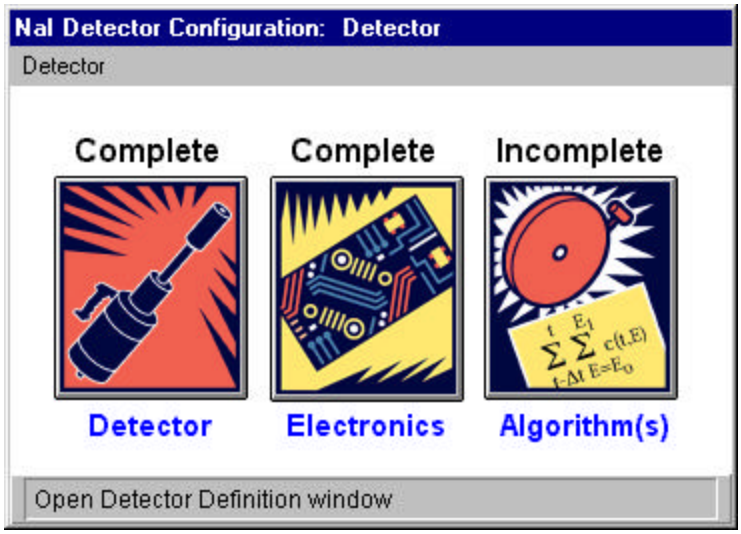


Figure 2.16: Detector configuration window.

Selecting the *Detector(s)* button of the *Detector Path Assignment* window brings up the *Detector Configuration* window. The *Detector Configuration* window provides access to the three subcomponents that make-up every detector. Displayed in the title-bar of this window is the name of the detector currently being manipulated. The *Window* pull-down menu provides the option of closing the *Detector Configuration* window. The specification of a detector is divided into three parts: detector definition, electronics, and algorithms.

Detector Definition

Selecting the *Detector* button of the *Detector Configuration* window brings up the *Detector Definition* window (Figure 2.17 and 2.18).

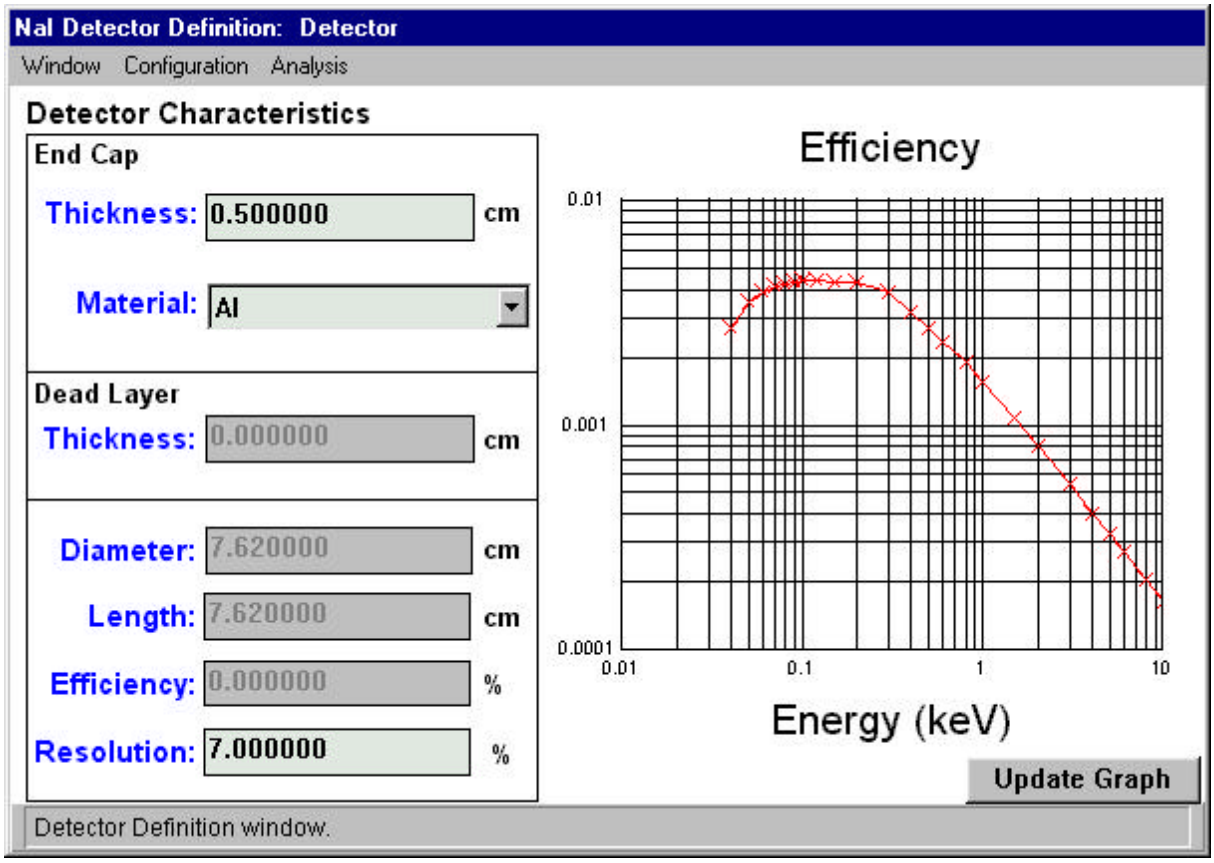


Figure 2.17: Detector definition window, NaI detector example.

Figure 2.17 depicts an example NaI detector configuration, and Figure 2.18 depicts a Ge detector configuration. Since Ge detectors are parameterized for physical characteristics such as size, length, etc., these values may be directly manipulated. However, only fixed NaI detector sizes are supported by QUEST v1, therefore these fields only display the default values for the specified NaI detector and are unavailable for data entry.

Other relevant detector parameters are also entered here and are used to correct the detector efficiency. For instance, the detector dead-layer thickness for germanium detectors can be specified, which allows the user to evaluate the difference between an N-type (no external dead layer) and a P-type (0.5 to 1.0 mm external dead layer) detectors. The effect of end-cap material is also included.

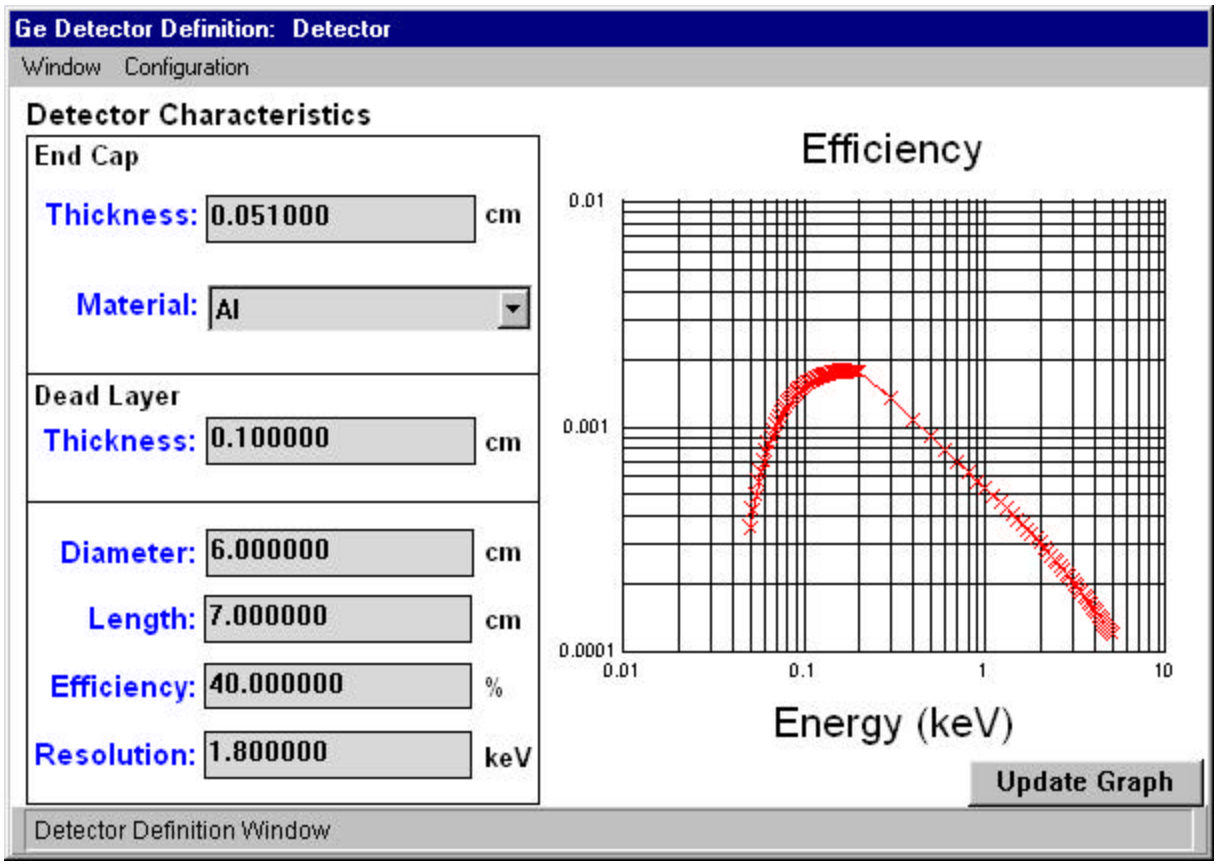


Figure 2.18: Detector definition window, Ge detector example.

Detector Electronics

Selecting the *Electronics* button from the *Detector Configuration* window brings up the detector's *Electronics Definition* window (Figure 2.19). The *Electronics* window allows the user to configure the simulated detector to mimic the hardware choices a user might make in a laboratory. The hardware choices include zero, gain, and a non-linearity term of the output-energy calibration. In addition, the number of data channels for the Analog-to-Digital Converter (ADC) can also be specified. The number of detector channels can greatly influence the real-time performance of the simulation, and so should be selected carefully. Displayed at the bottom of the window is the *Energy Scale*, which displays the minimum and maximum energies over the specified number of detector channels.

The *Signal* pull-down menu portion of the *Electronics Definition* window provides for the selection of statistical noise to add to the generated detector spectra. This can be a useful feature when trying to match an actual laboratory collected spectral response. By default this option is disabled.

Two other values are specified in the *Electronics Definition* window that control the performance of the simulation of the detector. These include the *Refinement Level* and *Points Per Refinement.* The default values for these parameters, 0 and 1 respectively, were chosen to maximize real-time simulation performance at the price of model accuracy. For a more detailed treatment of simulation performance related parameters see Section 2.2.2 Performance Tuning.

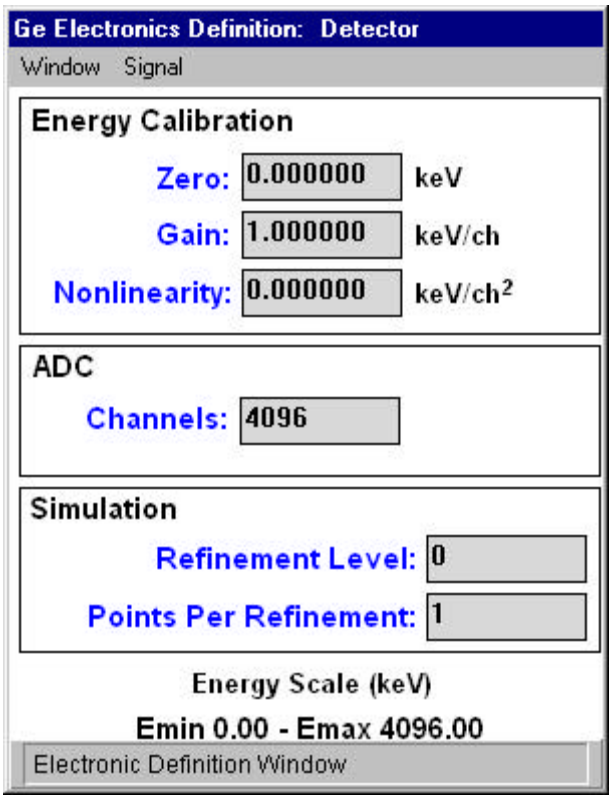


Figure 2.19: Detector electronics window.

Detector Algorithms

Each detector in a QUEST simulation may have an arbitrary number of associated algorithms. These algorithms are specified through selection of the *Algorithm(s)* button in the *Detector Configuration* window. Once selected, the Detector *Energy Definition* window appears (Figure 2.20). This window contains two tables, one for energy window specifications, and another for sample interval and algorithm name associations. For each specified sample interval and algorithm name association, the user can specify an arbitrary number of energy windows. These energy windows provide ranges over collected spectrum energy on which specific operations can be performed in calculating algorithm response. To specify an energy window, the user must enter the minimum energy, E_{min}, and maximum energy, E_{max}, in keV. Once these values are specified, pressing the *Insert* button will include the specified energy window into the energy window table located in the upper right-hand corner of the window. An energy window specification can be removed by highlighting a specific energy window specification in the table, and clicking the *Remove* button. Once inserted, an algorithm specification can be edited through selection of the *Define/Edit* button.

Clicking the *Define/Edit* button for a specific sample interval and algorithm name association brings up the *Algorithm Definition* window (Figure 2.21). It is from within this window that the user can utilize the previously specified energy windows and sample interval specification to modify the function of the associated algorithm.

Once all fields are specified, the user can check the completeness of an algorithm specification through selection of the "Check" button. If all values are consistent and within allowable ranges, the algorithm definition is accepted and the user is returned to the *Energy and Algorithms* window. If, however, there is problem with the algorithm specification, the user is warned of the problem, and given an opportunity to correct it.

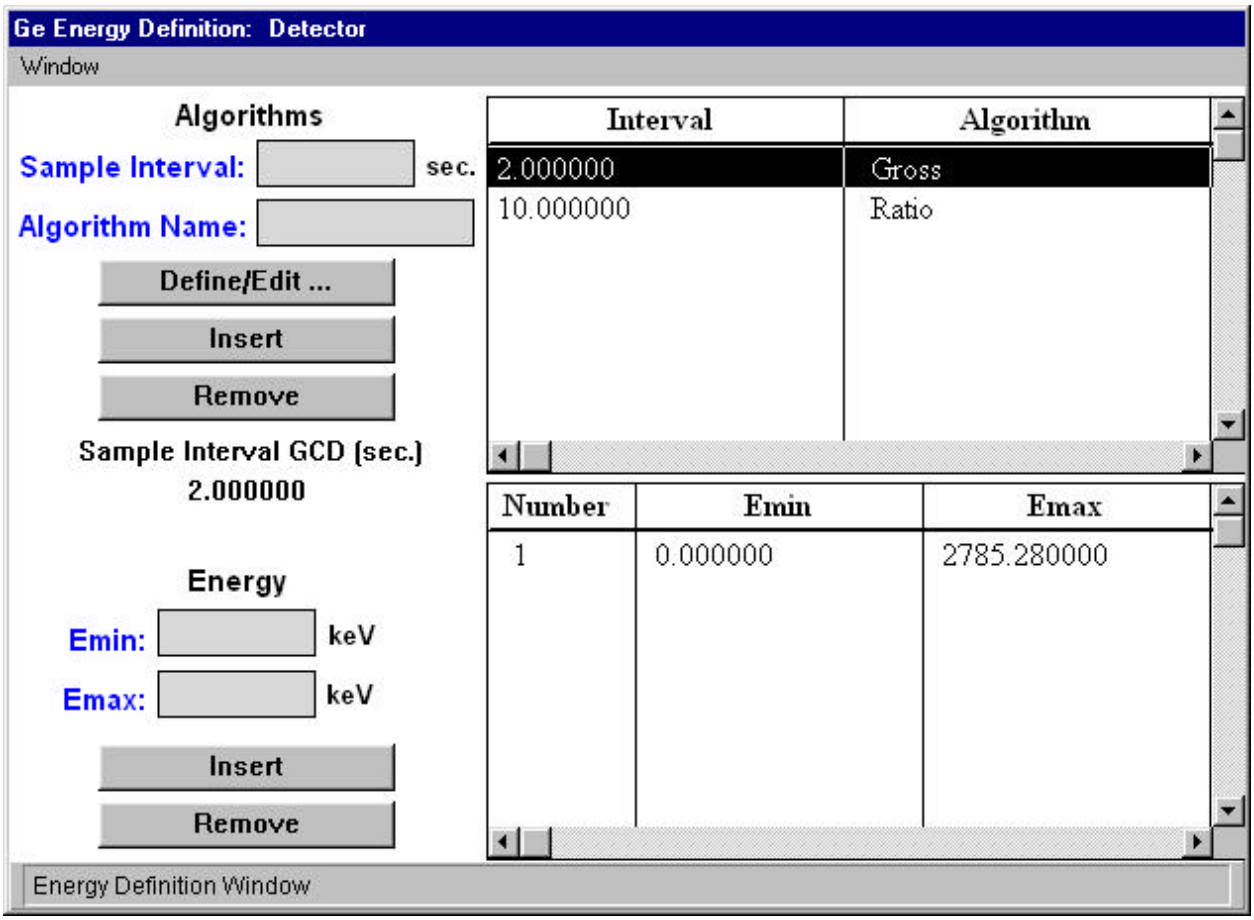


Figure 2.20: Detector energy definition window.

As sample interval and algorithm name associations are entered for the detector, the Energy and Algorithms window continuously calculates and updates the value of the Sample Interval Greatest Common Divisor (GCD), which is also the Least Common Factor (LCF). The LCF represents the least common factor of all specified sample intervals; this value is used as the iteration loop over which the spectral response for the detector is calculated. This has ramifications for performance of the model, and is provided as a user hint (see Section 2.2.2 Performance Tuning).

The detector algorithm logic can simulate a wide variety of detector algorithms with different settings of its parameters. Each simulated detector collects data using one or more sample intervals, and for each of its sample intervals a detector may use one or more algorithms. Only one algorithm's response is displayed during a simulation run, but the user may switch between different algorithms. The user may choose prior to the start of the simulation run to save the simulation history data. The responses of all the algorithms for all detectors are then saved to disk for later analysis (see Section 2.3 Analysis Mode).

The first set of parameters divides the energy spectrum into non-overlapping windows by specifying the minimum and maximum energy for each window. There may be any number of such windows. The user may group the energy windows into up to four window sets, W_1, W_2, W_3 and W_4. (For the algorithm to show any response at all, at least W_1 must be defined.) The detector represents the spectrum as counts in equally-spaced energy bins. The algorithm logic forms sums of the counts in the bins that fall into each window set; call them K_1, K_2, K_3 and K_4.

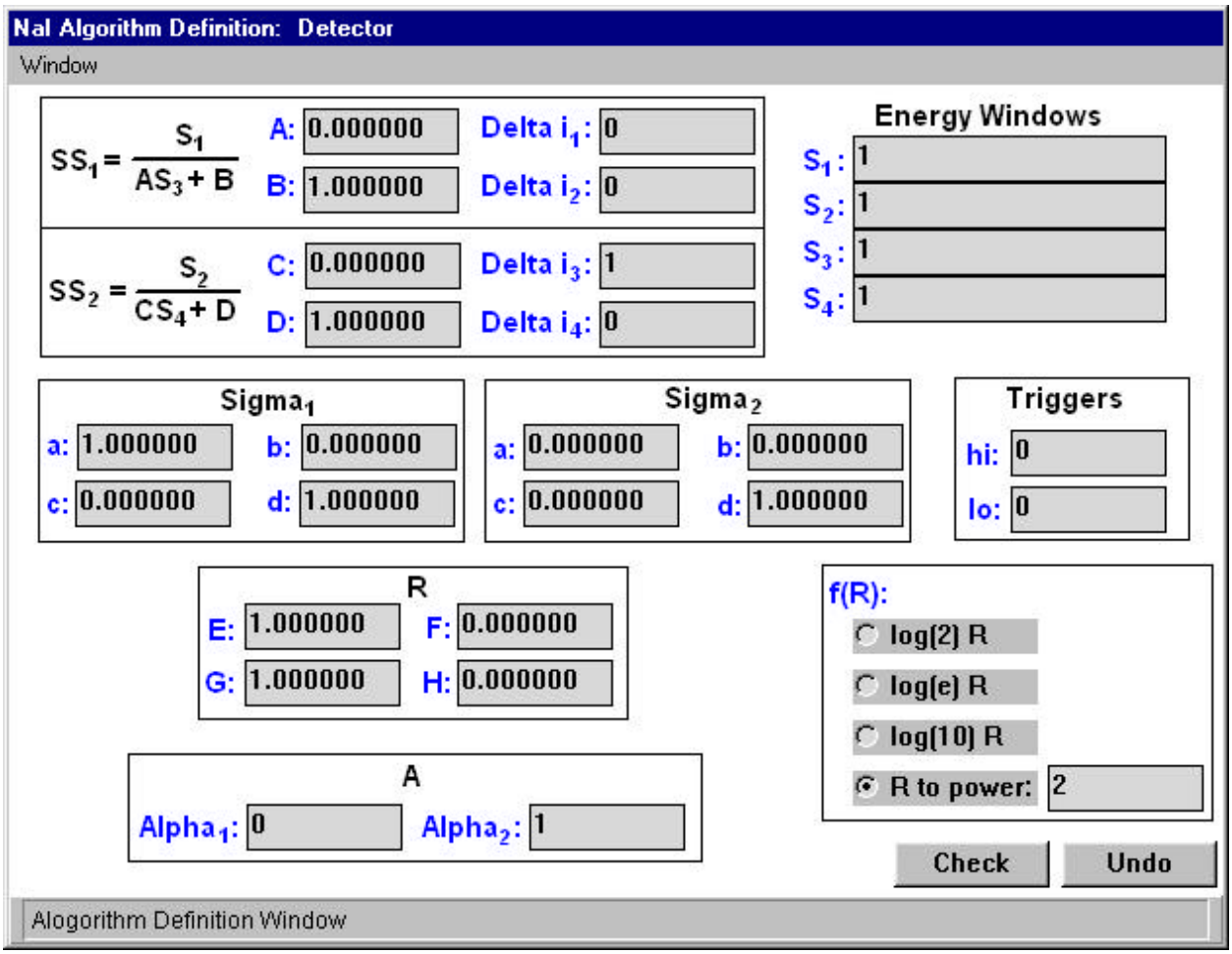


Figure 2.21: Algorithm definition window.

By setting the values of parameters *A*, *B*, *C* and *D*, the user can choose to work with counts or ratios of counts. The sums K_j are replaced by K_{ab} and K_{cd}, given by

$$K_{ab} = K_1 / (A * K_3 + B) \quad \text{and} \quad K_{cd} = K_2 / (C * K_4 + D).$$

If ratios are not to be used, K_3 and K_4 are not needed, hence W_3 and W_4 need not be defined.

Next K_{ab} and K_{cd} are replaced by averaged sums SS_1 and SS_2, given by

$$SS_1 = \frac{\sum_{k=k_0-\Delta_1}^{k_0-\Delta_2} K_{ab}(k)}{\Delta_1 - \Delta_2 + 1} \quad \text{and} \quad SS_2 = \frac{\sum_{k=k_0-\Delta_3}^{k_0-\Delta_4} K_{cd}(k)}{\Delta_3 - \Delta_4 + 1}$$

where $K_{ab}(k)$ and $K_{cd}(k)$ are the sums developed from the current detector sample if $k = k_0$, from the previous sample if $k = k_0 - 1$ and so forth. How many successive sums are averaged and which sums are included are controlled by setting the values of the parameters Δ_1, Δ_2, Δ_3 and Δ_4.

Next, the user has the option of introducing root-mean-square values by setting the parameters a_1, b_1, c_1, d_1 and a_2, b_2, c_2, d_2. Define

$$s_1 = \sqrt{\frac{a_1 \cdot SS_1 + b_1 \cdot \sum_{k=k_0-\Delta_1}^{k_0-\Delta_2} (SS_1 - K_{ab}(k))^2}{c_1 \cdot (\Delta_1 - \Delta_2) + d_1}} \quad \text{and}$$

$$s_2 = \sqrt{\frac{a_2 \cdot SS_2 + b_2 \cdot \sum_{k=k_0-\Delta_3}^{k_0-\Delta_4} (SS_2 - K_{cd}(k))^2}{c_2 \cdot (\Delta_3 - \Delta_4) + d_2}}$$

The input to the response function is formed by combining SS_1, SS_2, s_1 and s_2 according to the values of the parameters *E*, *F*, *G* and *H*:

$$R = \frac{E \cdot SS_1 + F \cdot SS_2}{G \cdot s_1 + H \cdot s_2}$$

The response function has the form

$$a_1 + a_2 \cdot f(R)$$

where $f(x)$ can be chosen to be $\log_b x$ (b = 2, e or 10) or x^p, where p is a positive integer. The coefficients a_1 and a_2, the logarithm base b and the power p are all specified by the user. The returned response value is clamped so that it never falls below zero. Also, when the logarithm function is chosen, the value returned by f is truncated to an integer.

SS_1 and s_1 are recomputed every time the detector develops a sample. The recomputation of SS_2 and s_2, however, can be made dependent on a threshold condition through the parameters *hi* and *lo*. After it computes SS_1 and s_1, the algorithm logic applies a high and low threshold test; if either is satisfied, SS_2 and s_2 are recomputed, otherwise they retain their former values. To satisfy the high threshold test, SS_1 must exceed ($SS_2 + H \cdot s_2$) for *hi* consecutive samples. To satisfy the low threshold test, SS_1 must fall below ($SS_2 - H \cdot s_2$) for *lo* consecutive samples.

As an example, the parameters for a true gross count algorithm will be presented. Give the parameters the following values:

$$\Delta_1 = 0,\ \Delta_2 = 0,\ \Delta_3 = 0,\ \Delta_4 = 0$$

$$A = 0,\ B = 1,\ C = 0,\ D = 1$$

$$a_1 = 1,\ b_1 = 0,\ c_1 = 0,\ d_1 = 1$$

$$a_2 = 0,\ b_2 = 0,\ c_2 = 0,\ d_2 = 1$$

$$E = 1,\ F = 0,\ G = 1,\ H = 0$$

$$a_1 = 0,\ a_2 = 1$$

$$f(x) = x^2$$

$$hi = 0,\ lo = 0$$

Let there be one energy window defined to cover the entire spectrum, and let energy window set W_1 be defined to contain that window. Then the sum K_{ab} developed from the current sample is the sum of all counts over the entire

spectrum. SS_1 is an average containing only the current sample value, hence is equal to K_{ab}. The sum in the expression for s_1 drops out since b_1 is zero, and we have

$$s_1 = \sqrt{K_{ab}}\ \text{, which leads to } R = \frac{1 \cdot K_{ab} + 0}{1 \cdot \sqrt{K_{ab}} + 0} = \sqrt{K_{ab}}\ .$$

Then the response value becomes $0 + 1 \cdot R^2 = K_{ab}$, which is the total number of counts over the entire spectrum, as desired. Once all algorithm definitions have been completed, the user can exit the Energy and Algorithms window through selection of "Close" through the Detector pull-down menu. It should be noted, however, that there must be at least one valid algorithm specified for each detector.

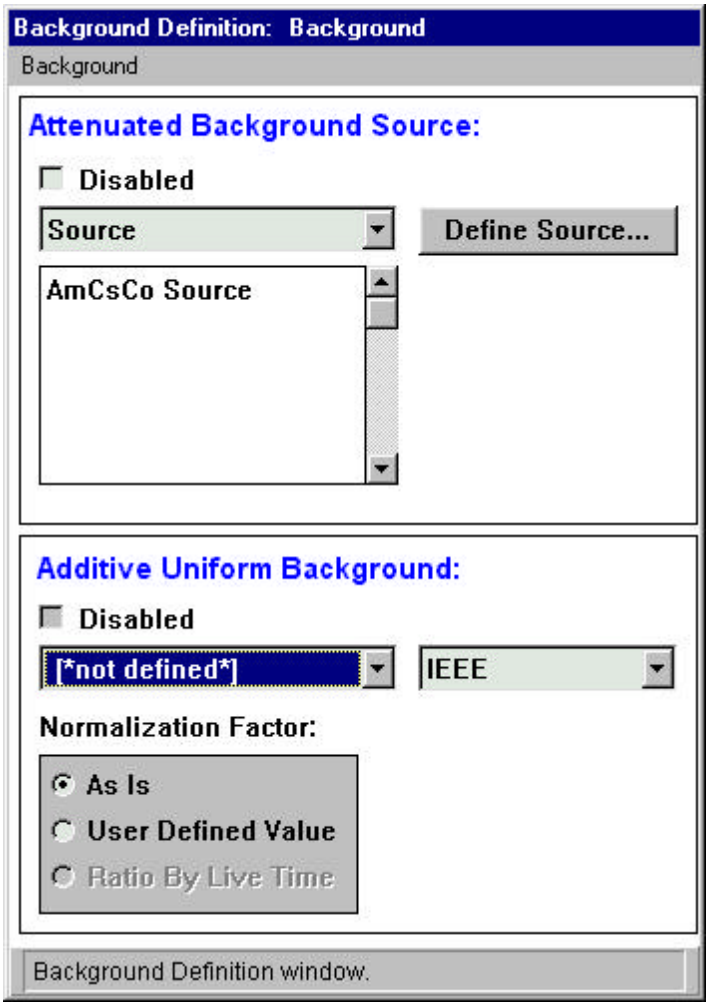


Figure 2.22: Background definition window.

2.1.5 BACKGROUND

Selecting the *Background* button from the *Primary QUEST* window brings up the *Background Definition* window (Figure 2.22). The background feature of QUEST may be used to add the effects of environmental radiation to the detected signal. Background is specified as a *source*, or a *spectrum*, or both.

A background source is a regular QUEST source, defined like other QUEST sources. Instead of being placed on a particular path, however, the background source receives special treatment to make it appear ubiquitous.

Any spectrum file in standard IEEE or ADCAM format may be used as a background spectrum. A background spectrum is simply added to the spectrum developed by the normal operation of the detector. Ideally, the spectrum should have the same zero, gain and number of channels as the detector; if it does not, it is interpolated to fit the detector.

2.2 SIMULATION MODE

Once a scenario has been defined or loaded, it can be simulated through selection of the *Simulation* button on the *Primary QUEST* window. Doing so brings up two additional windows, the *Simulation* (Figure 2.23) and *Simulation Response* (Figure 2.24) windows. During execution of a simulation, simulation output can be saved to disk file for later analysis. Immediately prior to beginning a simulation, a decision must be made to save output data or not.

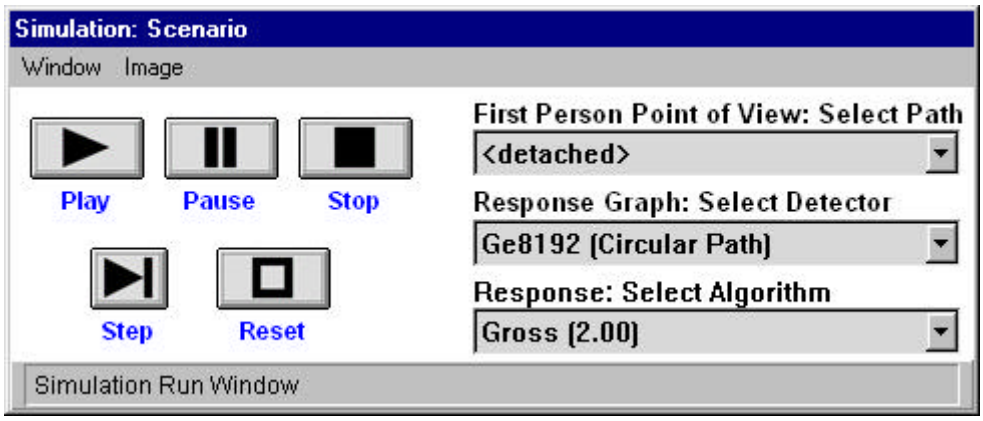


Figure 2.23: Simulation window.

2.2.1 SIMULATION CONTROL

The *Simulation* window contains "VCR-type" controls for managing the progression of the simulation. Listed in the title bar of the *Simulation* window is the name of the scenario being simulated. The right side of the *Simulation* window displays three pull-down selection boxes used in controlling the simulation. The *First Person Point of View: Select Path* selection box governs the path on which the first-person point-of-view (POV) tracks in the primary 3D graphics display window during progression of the simulation. The user may select any one of the defined paths. If a detached path is defined for the simulation, and the user switches the POV away from the detached path, the interactive, detached path remains stationary at that point until the POV is switched back.

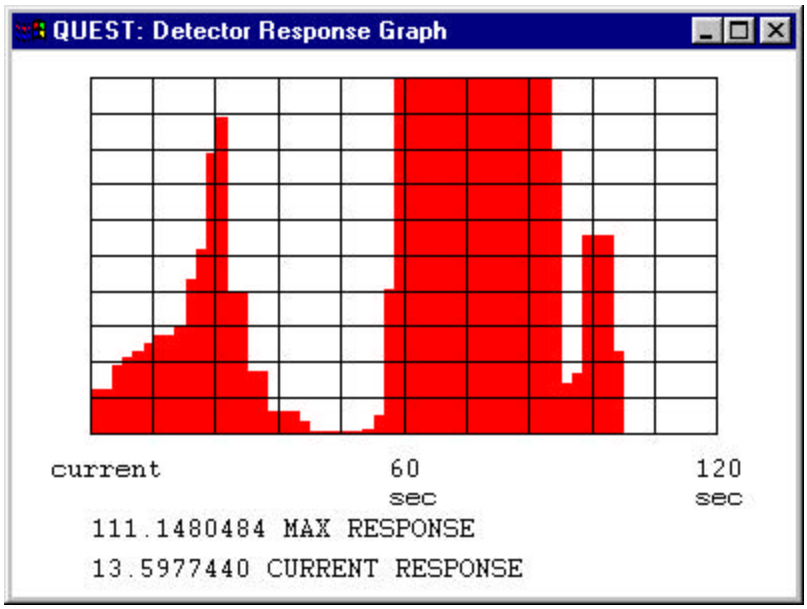


Figure 2.24: Detector response graph window.

The *Response Graph: Select Detector* and *Response: Select Algorithm* selection box determines the algorithm and detector for which the response chart displays output. Due to the computational demands of the simulation, only one detector algorithm can be displayed at a time during a simulation. The *Response Graph* displays the current algorithm response at the y-axis, with the history streaming off to the right.

The simulation continues until the user presses the "Stop" control button on the *Simulation* window. Once the button is depressed, the user is warned that the simulation is about to be terminated. If the user chooses to continue, all simulation output data is flushed to disk file and the simulation is halted. The user is then returned to the *QUEST Main Screen*.

In addition to the provided graphical user interface controls, QUEST responds to keyboard keystrokes. Pressing the 'u' key while the cursor focus is in the overhead-view graphics window will zoom-up the view. Likewise, pressing the 'd' key in the overhead-view graphics window will zoom-down the view. In general, when operating a QUEST simulation, the system performance is maximized by adhering to the following observations:

- Make the two graphics windows as small as possible.
- Prevent windows from overlaying each other (e.g. graphics windows and GUI control window).
- Prevent the 3D graphics windows from being "clipped" by the outside edge of the display.
- During a simulation keep the cursor focus in a 3D graphics window.

2.2.2 PERFORMANCE TUNING

QUEST attempts to address two application domains at the same time. On the one hand, QUEST performs well as a real-time simulation tool, providing interactive, human-in-the-loop simulations of the search for nuclear material. At the same time, however, QUEST is capable of providing detailed spectral data for specific radionuclide source, detector and absorber configurations. QUEST unifies these two application domains under a single simulation model, providing the user with the settings to manage the trade-off in application domain performance requirements. Thus, QUEST was designed with the capability to make highly accurate calculations when simulating detector responses. Unfortunately, higher accuracy requires increased QUEST execution time. The user interface provides a set of modifiable parameters for controlling the accuracy and detail of QUEST's transport physics computations—indirectly, this also controls real-time behavior.

QUEST must be capable of managing the computational demands of the four primary software components that comprise the application—these include the 3D graphics display (GRE) of the virtual environment, the mathematical model of the structural environment and the relationship between its components (CDB), the high computational demands of the transport physics' (TP) calculations, and the responsiveness of the graphical user interface (GUI). It should be obvious that, given a limited resource of a machine's computational capability, the requirements of these software components must be arbitrated based on the user's operational demand. For example, in order to support a real-time, human-in-the-loop simulation of the search for nuclear material (i.e. the use of QUEST as a searcher training tool), an emphasis would be given to the requirements of the 3D graphics engine. In contrast, to support trade-off studies in the design and use of radiation detectors (i.e. the use of QUEST as a analytic design tool), an emphasis would be given to the requirements of the transport physics. Thus, these trade-offs depend on the intended use of the application, and for each run, the user must decide best how to make these performance trade-offs between what is essentially computational accuracy and system responsiveness.

The user is given control of these performance trade-offs through access to various tuning parameters implemented in the application. The following bullets detail the major performance tuning parameters at the disposal of the user, their effects, and location within the simulation's GUI.

- Complexity of the structural environment. The more polygons the 3D graphics engine must render, the higher the computational demands.

- Number and complexity of sources simulated. Obviously, the greater the number of sources simulated, the greater the demands on the 3D graphics engine to render their movement, and the gamma-ray transport requirements imposed by the additional radionuclides. In the later case, the complexity of simulating a radioactive source is proportional to the number of gamma and x-ray lines associated with that source.

  Control: Obviously the user has control over the number of sources included in any given scenario. However, for each included source, the user should carefully manage the number of gamma and x-ray lines included. The *Radionuclide* window provides a ratio factor with which the user can eliminate all but the most energetic lines.

- Number and complexity of detectors simulated.

  Control: Once again, the user has direct control over the number of detectors include in a scenario. For each included detector, the complexity of device simulation can be greatly reduced by limiting the number of channels the detector collects samples over, the number of algorithms calculated (see Section 2.1.4), and the LCF of the defined detector sample intervals.

  This last point deserves greater explanation. Since spectral data for a detector is collected based on the sample interval defined sample-and-hold-period, the transport physics calculates the detector spectra for only one sample interval, regardless of the number defined for a specific detector. As discussed in Detector Algorithms of Section 2.1.4, this single sample interval is the LCF of all defined sample intervals. The smaller this value is, the greater the computational requirements.

- Simulation data collection and storage to disk requirements. During the execution of a simulation scenario, the user is only provided with the output of one detector algorithm at a time. In order to provide the more detailed post-processing capabilities of the Analysis Mode, simulation data must be collected and stored to disk. Wholesale collection and storage of simulation data is computationally expensive.

  Control: Under the *Detector Electronics* window, the user is given the choice of whether simulation data should be stored to disk or not.

◦ Transport physics accuracy requirements. Simulation of gamma and x-ray transport involves many complex levels of data integration over time and space. QUEST was designed with parameter controls on these calculations to allow the user to make the trade-off between result accuracy and model fidelity. The transport physics component of QUEST computes a response for every channel of every detector that accounts for all emitted source photons. Thus, execution time is roughly proportional to the number of detectors, the number of channels in each detector, and the number of distinct photon energies emitted by all sources. The number of detectors is fixed for a given simulation scenario, but the other two quantities can be altered to influence execution time. The parameters available to the user are:

Controls: Through the *Detector Electronics* window, the user has control over both the *Refinement Level* and *Points per Refinement*. QUEST simulates the sample and hold behavior of a detector by numerical integration over time. Each detector sampling period is covered by a number of equally spaced time samples. QUEST computes an instantaneous detector response at each time sample, then interpolates and integrates to get the total response. The number of time samples computed is determined by available CPU time; specifically, when the real-time clock advances beyond the hold time of a detector, then numerical integration stops. For example, if the detector's sampling period is 5 seconds and it takes 2 seconds for the computer to calculate the response at one time sample, then integration is based on only three sample points. This is a straightforward idea, but determining the optimum number of sample points is difficult when more than one detector is present. It turns out that responses at multiple time samples can be computed faster if they are "batched" together during one pass of transport physics calculations (one pass is also called one iteration of the "refinement loop"). The *Points per Refinement* parameter sets the maximum size of a "batch". For example, if the parameter is 2, then the detector computes a response at 2 time samples before examining the real-time clock. If a simulation is running slowly because some detector has a large number of channels, then setting the parameter to 1 should speed that detector up. Conversely, if a detector has very few channels, it might run faster with its parameter set to 3. *Refinement Level* defines the number of times the sample-and-hold period of the detector is divided down for integration. A value of zero indicates no division, and therefore only one large integration is performed over the entire sample-and-hold period. This results in better real-time response, but might cause problems when the sample interval is large relative to the detector's speed through the environment. Larger values for *Refinement Level* result in higher accuracy, but at the cost of real-time, interactive performance of the simulation. One consequence of the numerical time integration scheme is that execution time does not depend on the detector's sampling period; that is, performance cannot be improved by using shorter sampling periods.

Figure 2.25: Analysis control window.

2.3 ANALYSIS MODE

Once a simulation has generated output, it may be analyzed in more detail through the analysis mode. Analysis mode can be entered in two ways: through selection of the *Analysis* pull-down menu on the *Primary QUEST* window, or by pressing the *Analysis* button on the *Primary QUEST* window immediately following the run of a simulation. In either case, the *Analysis* control window (Figure 2.25) is displayed, along with one detector output window for each scenario specified detector.

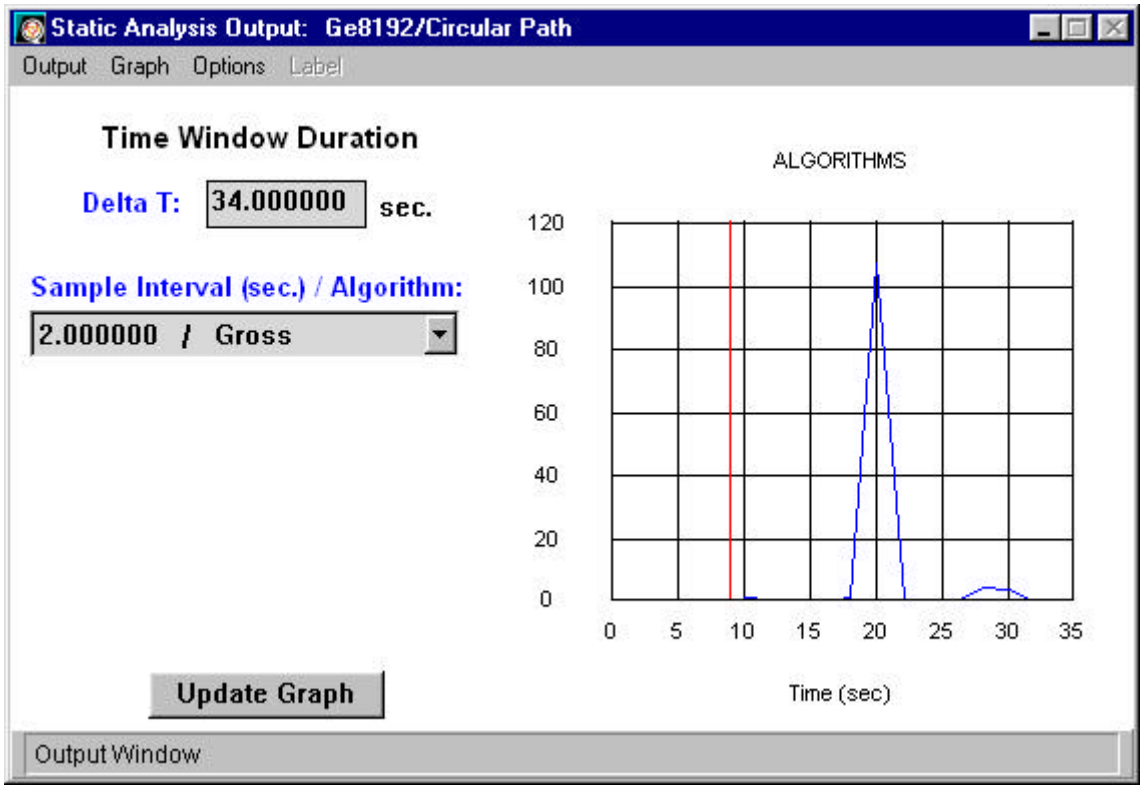


Figure 2.26: Static analysis output algorithm chart window.

The *Analysis Control* window provides playback controls for moving through the playback of data associated with the simulation. In addition, the *First Person Point of View* may be selected as any one of the scenario defined objects, sources or detectors, or an independent outside *Observer*. The *Duration* field lists the total amount of simulation time in seconds recorded in the output field. In addition, the *Time* field displays the current index of the output playback. This field may be directly edited.

For each detector defined in the scenario output file, a *Static Analysis Output* window is displayed (Figures 2.26 and 2.27). Two examples are given here to show the two types of output that are available for each detector, namely spectrum and algorithms. Figure 2.26 displays the algorithm output for a specified sample interval of the detector. The detector name for the output being displayed is given in the title bar of the window. The amount of algorithm time displayed can be changed with the *DeltaT* parameter, as can the algorithm being displayed.

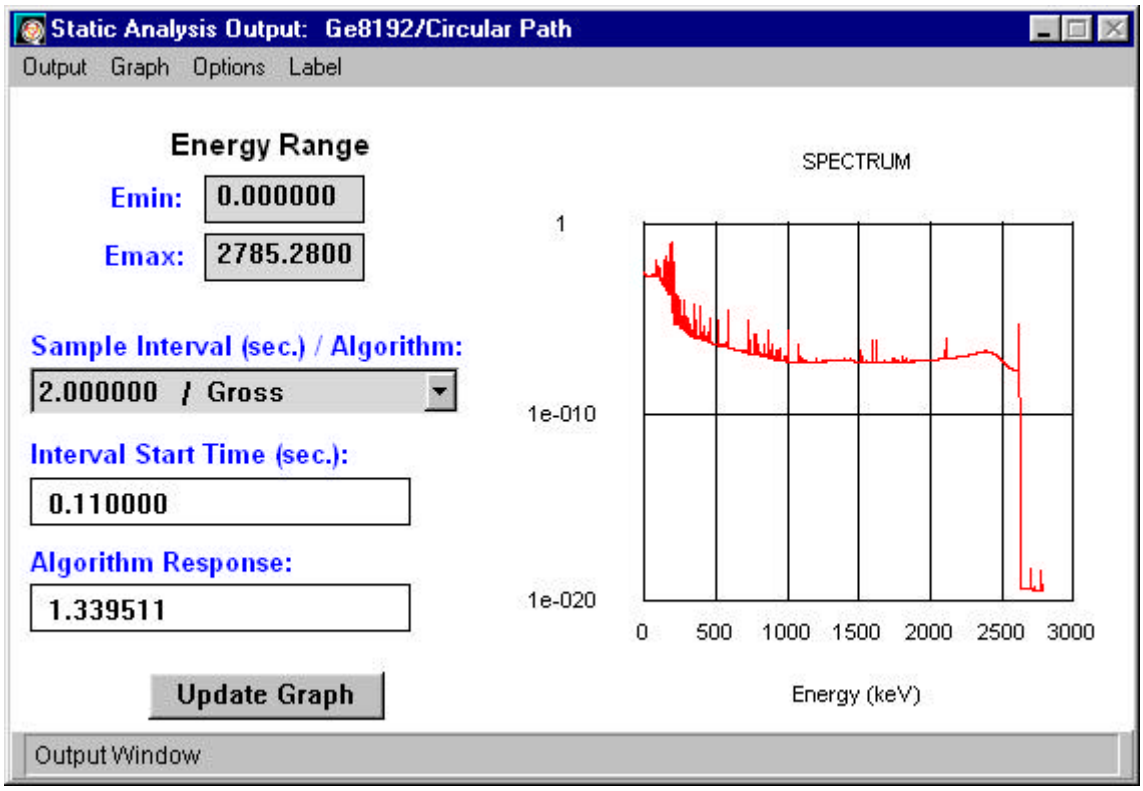


Figure 2.27: Static analysis output spectrum chart window

Figure 2.27 is an example of a *Static Analysis Output* window for a detector showing spectrum output. The energy range for the displayed spectrum can be varied through use of the *Emin* and *Emax* values. The *Interval Start Time* for the displayed spectrum is given, and one of the detector define *Sample Interval / Algorithm*s can be selected to display the *Algorithm Response*.

3 THEORY

QUEST is comprised of a number of interdependent software components, including: the Transport Physics (TP), Material Database (MDB), Background Radiation, Component Database (CDB), Interprocess Communication (IPC), Graphics Rendering Engine (GRE), and the Graphical User Interface (GUI) (Johnson et al. 1996). The TP, MDB, and Background Radiation components encapsulate radionuclide and gamma-ray databases, radiological source and detector models, as well as the radiation transport engine. The CDB provides the mathematical representation of the simulated environment supporting assignment of physical attributes (such as materials, thickness, color, etc.), and the physical relationships between structural model subcomponents. The IPC and GUI manage intercommunication between the software components and the visual representation of user information. And finally, the GRE maintains the three-dimensional graphic views, and supports interaction between the user and simulated environment, including real-time walk-throughs.

Each of these components encompasses a great deal of software detail. While it is beyond the scope of this paper to detail all design issues associated with QUEST, the following sections provide overviews of the theory underlying each software system, and provides an overview of their design and individual contribution to the simulation whole. The first section presents an overview, for the uninitiated, of the physics addressed by QUEST.

3.1 THE PHYSICS OF GAMMA RAY SCATTERING

As gamma rays pass through an object they interact with the particles that make up that object. These interactions result in an exponential decay in intensity of the gamma radiation, as may be seen through the following argument. Consider a thin slice of material containing targets with which the gamma ray can interact, as in Figure 3.1 (Delaney and Finch 1992). We model each of the targets in the material as a sphere with a cross sectional area called the interaction cross section, σ. A photon interacts with a target if and only if it passes through the corresponding sphere. The value of the cross section, σ, depends on both the energy of the incident photon and the nature of the target material. As will be seen in Section 3.1.2, many different types of interactions contribute to the cross section. However, to compute the total cross section we simply add the cross sections for each of the relevant interactions, i.e.,

$$s_{Total} = s_{interaction\,1} + s_{interaction\,2} + s_{interaction\,3} + \cdots \tag{3.1}$$

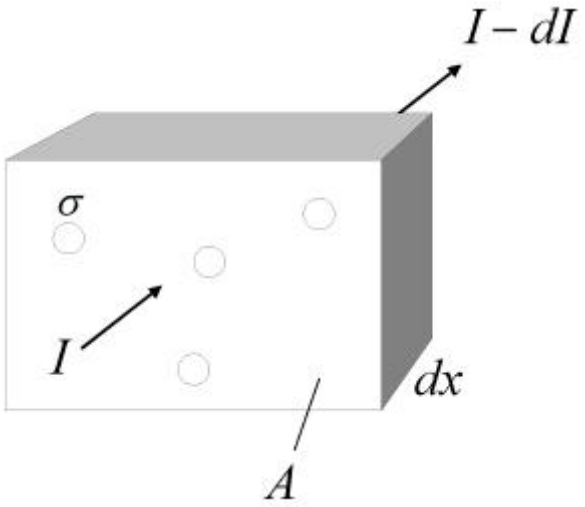


Figure 3.1: Gamma ray and material interaction.

The fraction of photons removed from the incident flux by the thin layer of material in Figure 3.1 is given by the ratio of the interaction cross sections of the targets to the total cross sectional area of the material

$$-\frac{dI}{I} = \frac{N(Adx)s}{A} = N\,s\,dx, \tag{3.2}$$

where I (the intensity) is the number of photons per second passing through a unit area normal to the path of the incident photons, dI is the number of these photons removed by interactions in the thin layer, and N is the number of target particles per unit volume. Integrating both sides of equation 3.2 gives

$$I = I_0 e^{-(Ns)x}, \tag{3.3}$$

where I_0 is the gamma ray intensity at x=0. If we introduce a new symbol m, called the attenuation coefficient, with $\mu = N\,s$, equation 3.3 becomes

$$I = I_0 e^{-mx}. \tag{3.4}$$

As for s, the total attenuation coefficient may be obtained by adding the attenuation coefficients for each of the relevant interactions, i.e.,

$$m_{Total} = m_{interaction\,1} + m_{interaction\,2} + m_{interaction\,3} + \cdots \tag{3.5}$$

Gamma ray photons interact with electrons, nuclei, and the electromagnetic fields associated with these particles. These interactions consist of either the scattering (i.e., change of direction and possible change in energy) or complete absorption of the incident photon (Table 3.1). Which of these effects gives the largest contribution to the attenuation coefficient depends on the energy of the gamma ray and the nature of the material through which the photon is passing.

Target	Scattering	Absorption
Atomic electrons	Compton scattering	Photoelectric effect
Nuclei	Nuclear scattering	Photonuclear reactions Photofission
Electromagnetic fields	Delbruck	Pair production

Table 3.1: Attenuation effects.

In the next section, we will describe each of the effects mentioned in Table 3.1 and discuss their relative contributions to the total attenuation coefficient for photon energies between 50 keV and 3 MeV. As we will see, the three most important interactions are Compton scattering by electrons (CS), the photoelectric effect (PE), and pair production (PP).

$$m_{Total} \approx m_{CS} + m_{PE} + m_{PP} \tag{3.6}$$

The photoelectric effect and pair production result in complete absorption of the photon and are both modeled by QUEST. In Compton scattering, however, the photon is not absorbed but, instead, changes direction and loses energy. Compton scattering thus effects the detector count rate in two ways. Photons that would have hit the detector may be scattered away from the detector by intervening materials, and photons that would not otherwise have hit the detector may be scattered into the detector.

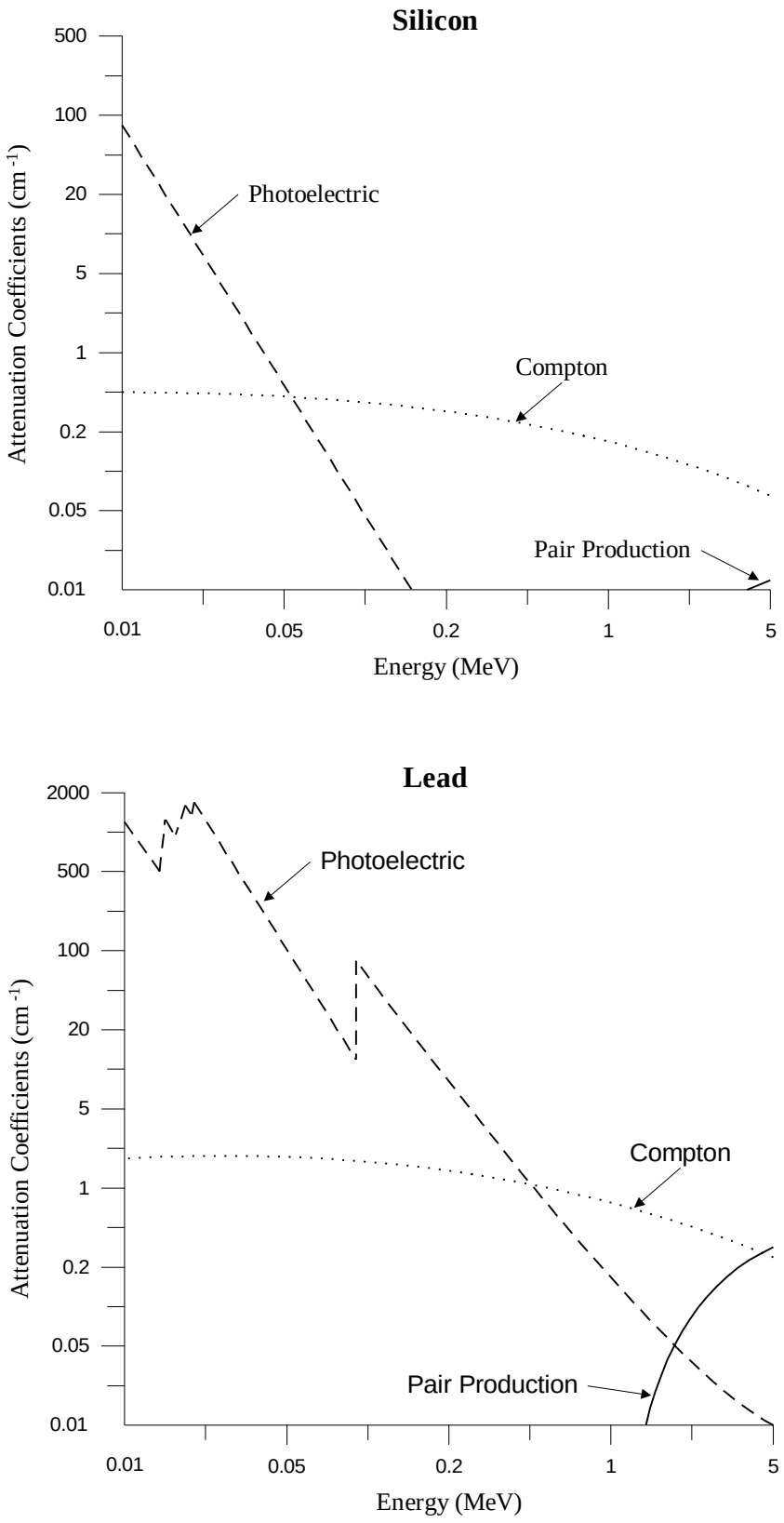


Figure 3.2: Attenuation coefficients versus photon energy.

The contribution of the first of these effects to the attenuation coefficient is computed in QUEST. However, the current version of QUEST does not include the second effect. This problem with Compton scattering will be discussed in more detail in Section 3.1.2.

3.1.1 ATTENUATION EFFECTS

From the graphs of attenuation coefficients versus photon energy in Figure 3.2, we see that Compton scattering gives the dominant contribution to the attenuation coefficient for intermediate photon energies. In addition, the range of gamma ray energies for which Compton scattering gives the largest contribution to the attenuation coefficient is greater for lighter elements, such as Silicon, than for heavier elements, such as Lead. The photoelectric effect dominates for low energy photons and heavier atoms, while pair production is only important for photon energies above, approximately, 2 MeV.

Compton Scattering

For photon energies much greater than electron binding energies, electrons may be treated as being essentially free. The scattering of photons by free particles is called Compton scattering. Since the contributions from Compton scattering for each electron in an atom add, the total contribution of atomic electrons to μ_{CS} is proportional to Z, the number of electrons in an atom. This weak dependence on Z, compared to the photoelectric effect and pair production, causes Compton scattering to be relatively less important for larger atoms. The attenuation coefficient for Compton scattering of photons by nuclei is less than 1% that for electrons since the attenuation coefficient for Compton scattering is proportional to one over the square of the mass of the scattering particle. For photon energies (E_g) above 100 keV, the attenuation coefficient is inversely proportional to E_g. This insures that pair production will dominate for energetic photons.

A significant complication introduced by Compton scattering is the scattering of photons into the detector. To see when this effect may be significant, consider the percentage of non-absorbed photons which Compton scatter when traversing various materials (Table 3.2).

Material	Thickness (cm)	Photon Energy (keV)	% of non-absorbed photons which Compton scatter
Al	0.5	200	15
		1000	8
		3000	4
concrete	10	200	98
		1000	87
		3000	68
air	2000	200	26
		1000	14
		3000	8

Table 3.2: Compton scattering of non-absorbed photons.

For gamma ray energies of interest (50 keV to 3 MeV) and relatively light building structures, the fraction of non-absorbed photons which Compton scatter is typically less than 25%. We would expect the change in count rate due to these photons hitting the detector to be small. However, for thick structures composed of heavier elements the fraction of non-absorbed photons which scatter approaches 100%. In this case we would expect these scattered photons to contribute significantly to the detector count rate under some conditions. To better model Compton scattering, future versions of QUEST may account for scattering into the detector.

Photoelectric Effect

Instead of scattering, a photon may be totally absorbed by an electron. If the gamma ray has more energy than the binding energy of the electron, the electron will be ejected from the atom in a process known as the photoelectric

effect. Interactions where the photon does not have enough energy to ionize the atom are less important since the photon energy must match exactly the difference in two atomic energy levels. The photoelectric effect really consists of two interactions: an electron absorbs a gamma ray and interacts with another electromagnetic field to conserve energy and momentum. This second electromagnetic field is typically the nuclear field. Thus, the stronger the nuclear electromagnetic field seen by an electron, the greater that electron's contribution to μ_{PE}. Inner shell electrons are in the strongest nuclear electromagnetic field and so give the largest contribution to μ_{PE} (provided the gamma ray energy is greater than these electrons' binding energy). The peaks on the graph of μ_{PE} versus E_g in Figure 3.2 are located where E_g becomes large enough to liberate the next closer shell of electrons. The electric potential seen by the innermost electrons may be written as

$$\text{Electric Potential} \sim \frac{\text{Nuclear Charge}}{\text{Size of Inner Electron Shell}} \sim \frac{Ze}{1/Z} \sim Z^2 \tag{3.7}$$

This potential depends on Z both through the nuclear charge and through the size of the inner electron shell. Thus μ_{PE} is highly dependent on Z ($\mu_{PE} \approx Z^4 - Z^5$) and the photoelectric effect will become increasingly important for materials containing heavier elements. As the incoming photon energy increases beyond the binding energy of the most tightly bound electrons, the electrons may be treated as being essentially free. Thus the photoelectric effect becomes less important while Compton scattering becomes relatively more important (see Figure 3.2.).

Pair Production

In pair production, a gamma ray splits into a particle antiparticle pair and one of these two new particles interacts with the electromagnetic field of the nucleus (usually) or another electron (less likely). The second interaction is necessary to conserve energy and momentum. To produce a particle antiparticle pair, the photon energy must be greater than or equal to the combined rest mass energies of the particle and antiparticle. For photon energies of interest here ($E_g \leq 3$ MeV), only electron positron pairs are light enough to be produced. The minimum gamma ray energy sufficient to produce an electron positron pair is

$$\text{Minimum } E_g = 2(\text{Rest Mass Energy of Electron}) = 1.02 \text{ MeV} \tag{3.8}$$

As E_g increases beyond 1.02 MeV, μ_{PP} increases as well. Since the contributions to the attenuation coefficient from the Compton and photoelectric effects decrease as E_g increases, pair production dominates for large E_g

As for the photoelectric effect, the stronger the nuclear electromagnetic field, the more likely pair production is to occur, although there is no dependence on the size of the innermost electron shell in this case. Thus the attenuation coefficient again depends on Z, though less strongly than for the photoelectric effect: μ_{PP} is proportional to Z^2.

Other Effects

The contributions of Compton scattering, the photoelectric effect, and pair production to the attenuation coefficient are all modeled by QUEST. As shown in Table 3.1, there are several other effects that may contribute as well. However, these effects are at most 1% of the first three and are ignored by QUEST. The error introduced by this approximation is small compared to the (typically ~10%) error caused by variations in building material compositions. Some of these additional effects and the reasons for their small size are discussed below.

First, consider reactions in which there are additional electron photon interactions. Examples are multi-photon Compton scattering, where there is more than one photon in the final state, and radiative corrections to all three of the effects discussed so far. (Radiative corrections to a reaction have the same initial and final states but involve additional photon interactions.) However, the extra electron photon interactions in these reactions reduce their contribution to the attenuation coefficient to at most 1% (for $E_\gamma \leq 3$ MeV) of the combined attenuation coefficients from Compton scattering, the photoelectric effect, and pair production—more precisely, the extra photon interactions will introduce additional factors of the fine structure constant, α, where $\alpha = 1/137$.

Instead of interacting with an electron, the incoming photon may react with a nucleus. As mentioned previously, the attenuation coefficient for Compton scattering of photons by nuclei is smaller than that for electrons by a factor of at least 100. Radiative corrections to this process are called Delbruck scattering. As for radiative corrections to Compton scattering by electrons, the attenuation coefficient for Delbruck scattering is further reduced from that for nuclear Compton scattering. In a process called photofission, a nucleus absorbs a photon and splits into two or more nuclei. Due to the large binding energies of most nuclei, photofission only gives a significant contribution to the attenuation coefficient for gamma ray energies above 5 MeV. Instead of splitting a nucleus, the absorption of a photon may cause transitions between nuclear energy states, i.e., photonuclear reactions. However, just as for transitions between atomic electron energy levels, the need to fine tune the photon energy to match exactly the difference in two (nuclear) energy levels reduces the importance of photonuclear reactions for most materials.

3.2 TRANSPORT PHYSICS

The Transport Physics (TP) is responsible for making calculations that simulate the emission and detection of radiation. A true radioactive sample emits gamma rays and X-rays (and possibly other particles) at specific energy levels. These photons may pass unimpeded through intervening materials, or possibly be absorbed, scattered, or reradiated before reaching a detector. Photons that pass into a detector are converted to electrical signals through a variety of physical processes, and these signals are processed to generate useful information. The most accurate model of the physics requires software simulation at the level of individual radiated photons; for example, Monte Carlo methods. However, the need to make calculations in a reasonably short time has led us to develop more approximate techniques, which are documented in the following sections.

3.2.1 ALGORITHM OVERVIEW

We choose to model only photon interactions occurring on a straight line between each source and detector. Scattering effects are approximated but treated as losses—no attempt is made to follow scattered or reradiated photons in the simulation. The mathematical models we use to represent physical processes are adaptations of those employed in the SYNTH software of Hensley (1994). However, an important design goal of QUEST is to structure the calculations in a manner that allows the accuracy of the simulation to improve with the amount of computer time available. This makes it easy to extend the QUEST software to more powerful processing architectures.

The physics simulation can be analyzed by considering the following topics:

- effects of moving sources and detectors,
- radiation losses while traveling between a source and detector crystal,
- conversion of intercepted radiation energy to electrical signals, and
- signal processing by a detector.

Moving Sources and Detectors

Sources and detectors in QUEST are allowed to move through space and change their axes of orientation. Thus, the photon flux received by a detector varies in time. Real detectors count photon interactions for a specified sampling period, then report the accumulated spectral information and move on to the next sampling period. The TP simulates sampling behavior by numerically integrating the received photon flux over time. Each detector carries out an integration specific to its sampling period and motion through space. An adaptive integration algorithm is used to allow tradeoffs between simulation accuracy and execution time.

Radiation Losses During Transmission

Given instantaneous source-detector geometry, the TP models photon emission along straight lines between each source and detector. Currently, detector crystals have finite spatial extent, but sources are treated as points. This means a straight-line path for simulated radiation can be any line segment starting from a source and ending somewhere inside the detector crystal. Each source emits a certain number of photons per second at specific energies. This photon flux is diminished before reaching the detector by collisions with other particles along the path. Since the final flux at the end of a path is geometry-dependent, the total flux received by the detector is a volume integral of individual flux contributions. We calculate the integral numerically using a special two-dimensional projection and an adaptive meshing technique borrowed from finite element methods. Again this allows the solution accuracy to increase with available computer time.

Converting Received Photons to Electrical Signals

A detector contains a crystal for converting received photons into electrons by a variety of physical processes, such as the photoelectric effect, Compton scattering, and pair production. In addition, the relatively narrow energy spectrum of a photon stream is spread out due to the thermal noise of the crystal atoms. Thus, a photon flux at one

energy is detected as a set of electrons with a continuum of energies. The TP models the electron-generating process for each incoming photon, and adds the individual continuums together to define a continuous energy function.

Detector Signal Processing

All detectors capture the energy continuum in a series of finite-width channels. As mentioned previously, they also integrate signals over a sampling period; thus, each reported channel is an integral over a range of energy and time. The user defines the energy range and number of detector channels. In addition, the TP models distortions due to nonlinearities in the electronic amplifier gain of the detector. The set of detected amplitudes in every channel is stored and fed into an "algorithm" defined by the user that reports a single number during the simulation.

3.2.2 Software Design Overview

Classes defined by the TP software are listed and described below.

LineSpectrum—An object of this class contains a set of discrete photon energies, each characterized by its flux rate and type of photon (X-ray or gamma ray). Two LineSpectrum objects from different sources can be merged into a single LineSpectrum.

EmittedRay—An object of this class describes the photon flux emitted from a point source along a ray to some other point in space. This object inherits photon descriptions from LineSpectrum. It uses the MDB to compute the attenuated photon flux delivered to the terminal point of the ray.

Path—An object of this class contains a collection of Node objects that define a portion of the recent trajectory of a source or detector through space. The trajectory can be thought of as a sequence of time-ordered points with an associated orientation vector (the axis of a cylindrical detector or anisotropic source). The object receives Node updates from the GRE and maintains them in time order. It can generate a point and orientation vector at arbitrary time values using linear interpolation. Only a finite number of the most recent Node updates are stored in memory; hence, requests for data must be confined to this time interval.

Detector—An object of this class defines the operating characteristics of a real detector device. Methods in this class provide the top-level driver for the TP algorithms, described later in this document. Briefly, a detector uses Path information to set up EmittedRay objects from all point sources. After computing the received LineSpectrum of photons, the detector models physical processes that smear the discrete spectrum into a ChannelSpectrum object. A sequence of time-ordered ChannelSpectrum objects is analyzed to generate a single scalar output from each detector.

ProjMapping—An object of this class describes an abstract coordinate mapping between three-dimensional Euclidean space and an embedded two-dimensional projection space. It is used to make coordinate transformations between the two systems.

ProjDetector—An object of this class represents the two-dimensional projection of a detector crystal as seen by a particular point source. Many such objects are created and destroyed during computations made by each detector. Each object inherits an associated ProjMapping object to describe the projection mapping. The primary function of a ProjDetector is to compute the cross-sectional area of a three-dimensional detector crystal as seen from a particular point source.

The following high level outline describes the operation of the TP. Its purpose is to indicate the flow of execution during a simulation.

```
Load Detector information for the current scenario
Wait until some Path contains enough Nodes to completely cover a detector's sampling period
```

```
while the simulation remains active
    for each Detector
        if simulation time > current sampling period end time + Δt_D
            then save final ChannelSpectrum and start on the next sampling period
        Choose the next set of time instants for integrating the current sampling period
        for each time instant
            loop over all Sources
                Compute cumulative LineSpectrum received by the Detector
            end loop
            Compute a ChannelSpectrum corresponding to this time instant
            Update the time integration for the current sampling period
        end for
    end for
end while
Deallocate memory, clean up in preparation for possible new scenario
```

3.2.3 Accounting for Source and Detector Motion

The QUEST software allows sources and detectors to move along paths, which are defined as a time-ordered sequence of points in space. In addition, the orientation of detectors (where they are pointed) and of anisotropic sources can change along the path. As a result, motion through a building with walls and doorways can cause the detected energy to vary suddenly along a path.

In practice, all real detectors count photons over a finite sampling period (typically about 10 seconds) and report the accumulated spectral information. We divide paths into segments whose duration equals one sampling period and compute the detector response for each segment. The accumulated response is fundamentally given by an integral over time, which we approximate using an adaptive integration algorithm.

Path Construction

The path of each object is obtained from the GRE as a time-ordered sequence of point and orientation vectors (also referred to as *nodes* of the path). The GRE delivers a batch of nodes at regular intervals, with one time stamp for the whole batch. The TP uses the time stamp of the current and previous batches to attach a time to each node. A list of time-tagged vector pairs is then stored for each object in an array of fixed length. The length is chosen to hold enough path information for two sampling periods of the slowest detector (exact determination of the array length also depends on how fast the GRE can generate data).

During a simulation each source and detector is associated with a Path object that contains its array of time-tagged position and orientation vectors. Various TP algorithms may request information from the Path at arbitrary times; it is the responsibility of the Path object to return meaningful data. This is done using simple linear interpolation between the two nearest nodes. The Path should never receive requests for data outside its storage range. Even though the GRE and TP operate as separate threads in QUEST, provision has been made to keep the two processes synchronized.

Numerical Integration

Each detector has one or more sampling periods, denoted as Δt_D. Multiple periods are allowed so the user can make comparative studies between detectors with different parameters. The TP creates a local Detector object for each possible sampling period, and these objects (each with a single Δt_D) compute responses independent of one another.

The sampling period of a detector defines a segment of time over which photons are to be counted. One can visualize a spatial path segment for the detector and all sources corresponding to this time interval—this is the geometry over which a response is calculated. However, computation of the response is not as difficult as this picture suggests. The detector simply queries each source for its position at certain discrete time instants and computes the instantaneous response. These responses are accumulated over the sampling period to provide a total photon count.

Discrete time instants are chosen using numerical integration with an adaptive width parameter. Integration of a new sampling period starts by choosing the time instant in the middle of the period. If the available computation time for the TP is used up in calculating the response at this time, then we multiply the response by Δt_D and quit. This approximates the time integral by a function value at a single point (the centroid of the sampling interval). If more computer time is available, then we choose the two ends of the sampling interval. Two more instantaneous responses are computed, allowing the integral to be more accurately approximated from three data points. The sampling interval can be further subdivided as computation time permits, providing ever greater accuracy.

Our numerical integration scheme uses the trapezoid method for computing approximate total responses. This has only first-order accuracy, but requires storage of just one extra instantaneous response (a ChannelSpectrum object, which can use as much as 100 Kbytes of memory). Integration can be terminated at any point in the algorithm to accommodate real time performance requirements. If computer time is readily available, then integration proceeds until the relative change in final response is smaller than a user-defined accuracy threshold.

Note that our adaptive integration scheme may query Path objects for position information at any time instant in the sampling period. Therefore, integration cannot begin until all GRE data for a sampling interval has been obtained. This means TP computations lag by one sampling period; i.e., results are displayed up to Δt_D seconds late.

3.2.4 Calculating Losses Between Source and Detector

The basic computation in this section is the determination of the photon flux (number of photons per second) that passes into a detector crystal at a given instant of time. This is found by computing the photon flux radiated from all sources and integrating over the finite detector crystal volume. Each source is a point that emits a discrete line spectrum of photon energies in all directions according to some radiation pattern. We simulate photons that follow a straight line to some portion of the detector crystal, computing absorption and scattering losses to intervening materials, but we ignore photons that might reach the detector by scattering off intervening particles. The detector volume subtends a certain solid angle with respect to each radiating source, tracing out an irregularly shaped cone. The contribution of each source is given by a volume integral of photon flux computed over the set of rays within its subtended cone. The TP computes one integral for each source, but the solid angle is replaced by a simpler two-dimensional geometry. The simple algorithm below summarizes our computations.

```
for each Detector
    loop over all Sources
        compute the two-dimensional projected detector region seen by the source
        loop over mesh points on the projected detector
             initialize a LineSpectrum and EmittedRay from source to mesh point
             multiply by the source anisotropic radiation factor
             compute attenuation due to intervening materials
             add the mesh point contribution to the photon flux integral
        end loop
        scale the integral to restore three-dimensional perspective
    end loop
end for
```

Projecting the Detector

Each source emits radiation that is intercepted by the detector crystal. The TP models a source as a point and computes the total flux radiated into the detector by adding up contributions from the cone of rays connecting the source and detector. Instead of integrating over this solid angle, we use a two-dimensional projection of the detector crystal onto a flat plane. The projection plane passes through the detector centroid and is perpendicular to the line between the source and detector centroid (a more proper projection would be onto the surface of a sphere centered at the source and passing through the centroid, but the flat projection we use is far simpler to model). We integrate radiation received over this two-dimensional region numerically using a triangular mesh. Each triangle vertex defines an EmittedRay object coming from the source, and we calculate the photon flux along this ray. The approximated integral is simply a linear combination of values at the vertices.

To describe the projection operation, define a Cartesian coordinate system with the point source at the origin, the centroid of the detector along the positive y-axis, and the axis of the cylindrical detector in the xy-plane. (This can always be transformed back to the coordinate system of the original problem using translation and rotation operators that leave areas invariant.) Then the plane we project onto is defined by the equation $y = \bar{y}$, where $\bar{y}$ is the distance between the source and detector. The projection of any point (with positive y-coordinate) is found by extending a ray from the origin through the point and finding where it intersects the projection plane; thus, (x, y, z) projects into the point $(a\,x, \bar{y}, a\,z)$, where $a = \bar{y}/y$. We can use this formula to deduce an analytic expression for the projection of a detector in the shape of a right circular cylinder.

Let the cylinder have diameter d and height h. Let its axis, which we specified to be in the xy-plane, make an angle q with the projection plane. Making the restriction $0 \le q \le p$, we see that the two endpoints of the cylinder's axis are located at

$$(\frac{h}{2}\cos q, \bar{y} + \frac{h}{2}\sin q, 0) \quad \text{and} \quad (-\frac{h}{2}\cos q, \bar{y} - \frac{h}{2}\sin q, 0).$$

Each of these points is the center of a circular base of the cylinder. Projecting the circular outline of each base gives an ellipse in the projection plane. If we connect the two ellipses by tangent line segments we will have the outline of the projected detector, which is the two-dimensional region to integrate over. The two circular outlines are best expressed in terms of an angle parameter φ, which runs from 0 to 2π. For instance, the circle whose center is the first point given above has parameterization

$$x = \frac{d}{2}\sin q \cos j + \frac{h}{2}\cos q$$

$$y = \bar{y} - \frac{d}{2}\cos q \cos j + \frac{h}{2}\sin q$$

$$z = \frac{d}{2}\sin j \ .$$

Applying the projection formula will give a three-dimensional parameterization of the ellipse, which we know is confined to the projection plane $y = \bar{y}$. Let (u, v) be the two coordinates in the projection plane, where $u = x$ and $v = z$ whenever the y-coordinate is equal to $\bar{y}$. Then the projected ellipse, still parameterized in terms of φ, is

$$u = \frac{\bar{y}}{\bar{y} - \frac{d}{2}\cos q \cos j + \frac{h}{2}\sin q}\left(\frac{d}{2}\sin q \cos j + \frac{h}{2}\cos q\right)$$

$$v = \frac{\bar{y}}{\bar{y} - \frac{d}{2}\cos q \cos j + \frac{h}{2}\sin q}\left(\frac{d}{2}\sin q\right).$$

The next step is to eliminate φ using trigonometric identities, obtaining a quadratic equation in terms of the variables u and v. From this equation of the projected ellipse, we get formulas for its center and the lengths of its axes. Note that because we chose a coordinate system with the axis of the cylindrical detector in the xy-plane, the projected ellipses are rectilinear in the (u, v) coordinates. After some algebra, the answer for the first base center given above is

$$\text{ellipse center at} \quad u_{c1} = \bar{y}\cos q \, \frac{\frac{1}{4}(h^2 + d^2)\sin q + \frac{\bar{y}h}{2}}{(\bar{y} + \frac{h}{2}\sin q)^2 - (\frac{d}{2}\cos q)^2}$$

$$\text{semi-axis lengths} \quad u_{a1} = \frac{\bar{y}d}{2} \, \frac{\bar{y}\sin q + \frac{h}{2}}{(\bar{y} + \frac{h}{2}\sin q)^2 - (\frac{d}{2}\cos q)^2}$$

$$\text{and} \quad v_{a1} = \frac{\bar{y}d}{2} \, \frac{1}{\sqrt{(\bar{y} + \frac{h}{2}\sin q)^2 - (\frac{d}{2}\cos q)^2}}.$$

The second base center yields a similar projected ellipse characterized by

$$\text{ellipse center at} \quad u_{c2} = \bar{y}\cos q \, \frac{\frac{1}{4}(h^2 + d^2)\sin q - \frac{\bar{y}h}{2}}{(\bar{y} - \frac{h}{2}\sin q)^2 - (\frac{d}{2}\cos q)^2}$$

$$\text{semi-axis lengths} \quad u_{a2} = \frac{\bar{y}d}{2} \, \frac{\bar{y}\sin q - \frac{h}{2}}{(\bar{y} - \frac{h}{2}\sin q)^2 - (\frac{d}{2}\cos q)^2}$$

$$\text{and} \quad v_{a2} = \frac{\bar{y}d}{2} \, \frac{1}{\sqrt{(\bar{y} - \frac{h}{2}\sin q)^2 - (\frac{d}{2}\cos q)^2}}.$$

The two ellipses just calculated represent the projections of the circular ends of the cylindrical detector. To finish the outline of the projected detector, we must draw a line segment connecting the tops of the two ellipses, and a mirror-image line segment connecting the bottoms. The line segment must be tangent to each ellipse at the points where it intersects them. Finding the two points of intersection requires solving a system of four quadratic equations in four unknowns. Assuming that $u_{c1} < u_{c2}$, the equations are:

$$\left(\frac{u_L - u_{c1}}{u_{a1}}\right)^2 + \left(\frac{v_L}{v_{a1}}\right)^2 = 1$$

$$\left(\frac{u_R - u_{c2}}{u_{a2}}\right)^2 + \left(\frac{v_R}{v_{a2}}\right)^2 = 1$$

$$v_L(v_L - v_R) + \left(\frac{v_{a1}}{u_{a1}}\right)^2 (u_L - u_R)(u_L - u_{c1}) = 0$$

$$v_R(v_L - v_R) + \left(\frac{v_{a2}}{u_{a2}}\right)^2 (u_L - u_R)(u_R - u_{c2}) = 0,$$

where (u_L, v_L) and (u_R, v_R) are the left and right intersection points, respectively. The nonlinear system of equations is solved using Newton's method [Dennis], properly safeguarded to account for degenerate cases.

Centroid Integration Method

The two ellipses and connecting line segments completely define the outline of the two-dimensional projection of a cylindrical detector crystal. The region inside this outline corresponds to the region over which we integrate received radiation. QUEST is designed to support an adaptive triangular mesh for computing this integral; however, when running in real time a simpler "centroid method" of approximate integration is used. The centroid method computes received radiation at a single mesh point, the origin of the (u, v) coordinate system, and multiplies this by the area enclosed by the projected detector outline. Thus, it makes the simplifying assumption that the radiation flux is a constant over the whole detector.

Source Energy Spectrum

In the computations above we needed the photon energies that reached a particular mesh point on the detector from some source. These energies are expressed in the software as a LineSpectrum object, which consists of a set of discrete spectral lines. Each line's strength is initialized with the total number of decays/second made by the source at that energy. LineSpectrum objects are created for each source, taking into account the source's elemental composition and mass. The LineSpectrum is then affiliated with an EmittedRay vector that emanates from the source to a point on the detector. If the source radiates anisotropically, then the source strength is uniformly attenuated by a factor determined by the angle between the EmittedRay vector and the orientation axis of the source.

Flux Attenuation

As they travel from source to detector along an EmittedRay, radiated photons are absorbed or scattered by electrons in the nuclei of intervening materials. We calculate a linear attenuation coefficient μ for each type of material along a path, and reduce the average number of photons emitted per second by the factor $e^{-\mu d}$, where d is the path distance through the material. The linear attenuation coefficient is calculated as $\mu = \rho A_E$, where ρ is the density of the material (in grams per cubic centimeter), and A_E is the mass attenuation coefficient (in centimeters squared per gram).

The mass attenuation coefficient depends on the energy of the radiated photon and the atomic properties of the attenuating material. Following SYNTH, coefficients are computed using models developed from data tabulated by the U.S. Department of Health, Education, and Welfare in *Radiological Health Handbook* (1970). Attenuation losses from photoelectric, Compton, pair production, and K-shell effects are each computed from empirical formulas, then adjusted for the atomic weight of the absorbing element. Let E be the energy of a photon, Z the atomic number of an absorbing element, and A the atomic weight. Then,

$$A_E(E, Z, A) = (c_{ph} + c_{cmp} + c_{pr}) c_{Ksh} (A_{cal} / A),$$

where:

$$c_{ph} = \exp[6.029 - 0.6624(\ln Z) + 1.4478(\ln Z)^2 - 0.2033(\ln Z)^3] \left\{ \exp\left[\frac{-0.693 \ln(1000E)}{0.217 + 0.00055Z} \right] \right\}$$

$$c_{cmp} = (0.066 - 0.0003314Z + 0.00000277Z^2) \exp[-0.4457(\ln E) - 0.04707(\ln E)^2]$$

$$c_{pr} = \exp[-9.108 + 0.0685Z - 0.000395Z^2 + 1.231(\ln E)], \quad \text{if } E \geq 2e_m \quad (\text{otherwise } c_{pr} = 0)$$

$$\text{with} \quad e_m = 0.511006 \text{ MeV}$$

$$c_{Ksh} = 0.10266 + 0.006798\sqrt{E_K} + 0.0006539E_K, \quad \text{if } 1000E < E_K \quad (\text{otherwise } c_{Ksh} = 1)$$

$$\text{with} \quad E_K = -0.33078 + 0.021268Z + 0.008907Z^2 + 0.00004918Z^3$$

$$A_{cal} = -1.0364 + 2.1317Z + 0.0048504Z^2 .$$

Photon flux is attenuated according to the mass density of each species of element present in an intervening material. For example, passage through water causes an attenuation from both hydrogen and oxygen nuclei. A mass attenuation coefficient is calculated for each element, then multiplied by the density of the nuclei. The software stores the relative abundance of each element in a material as its fraction by weight; thus, the density of an element is simply its relative abundance times the density of the material. Continuing the example of water, the relative abundance of hydrogen atoms is computed from atomic weights to be

$$\frac{2(1.00797)}{2(1.00797)+15.9994} = 0.112 .$$

The density of hydrogen nuclei in water is therefore 0.112(1.00) = 0.112 grams per cubic centimeter.

Conversion to Solid Angle

Our two-dimensional approximate integration of radiation flux yields a quantity with the units decays/second times centimeters squared. The flux function accounts for source radiation anisotropy and attenuation losses to intervening materials, but uses the total emitted flux of the source sample. What we want is the fraction of total emitted radiation intercepted by the detector crystal; therefore, we need to divide by the total surface area of a sphere that is centered at the source and passes through the detector. If we had projected the detector onto the surface of this sphere instead of a flat plane, then this calculation would give the exact solid angle subtended by the detector. Our use of a flat plane introduces some small error, the worst case occurring when a detector and source are close together.

3.2.5 Modeling Detector Physics

The computations in this section convert a discrete LineSpectrum of received photon energies into a continuous spectrum of detected energies. Photons are converted to detectable electrons by the NaI or Ge material that makes up the detector crystal. The electrons generate an electrical signal that is then amplified, integrated, and processed as described in the next section. Here, we concentrate on the physics of detection in the crystal.

ρ	Density of a material ($^{gm}/_{cm^2}$)
A_E	Mass attenuation coefficient of an element (squared $^{cm}/_{gm}$)
μ	Linear attenuation coefficient of a material at a particular energy ($^{1}/_{cm}$)
E_γ	Single discrete energy representing a stream of gamma ray photons (MeV)
E_c	Compton energy, the upper limit of the Compton continuum (MeV)
m_e	Rest mass of an electron, 0.511006 MeV
$eff_{1.333}$	Efficiency of Ge detector (percent)
σ	Standard deviation of a Gaussian pulse (MeV)
$FWHM$	Full width of a pulse at half maximum values (MeV)
$f_p(E)$	Detector gamma ray response due to photoelectric effect ($^{counts}/_{sec}$)
$f_{pX}(E)$	Detector X-ray response due to photoelectric effect ($^{counts}/_{sec}$)
$f_t(E)$	Detector "tailing term" response due to photoelectric effect ($^{counts}/_{sec}$)
$f_c(E)$	Detector response due to Compton effect ($^{counts}/_{sec}$)
$f_{mc}(E)$	Detector response due to "multiple Compton" effect ($^{counts}/_{sec}$)
Δt_D	Electronic sample and hold period of a detector (sec)

Table 3.3: Symbols used to model detector physics.

Detectable electrons are created from high-energy photons by the photoelectric effect, Compton scattering, and pair production. In addition, the very narrow energy spectrum of a photon stream is spread out due to the thermal motion of the crystal atoms. In the remainder of this section we let E_γ denote the discrete energy of a source photon stream, and describe how to compute its detectable spectrum (symbol definitions are given in Table 3.3).

Photoelectric Effect

This occurs when a photon collides with a bound electron in the crystal, giving up all its energy to the electron, which is knocked out of its atom. The photoelectric effect is the dominant detection process, producing a peaked energy spectrum very close to the energy of an incoming photon stream. The fraction of photons converted by the photoelectric process is determined by the detector's efficiency at that energy.

The efficiency of Ge detectors is calculated using the curve fitting models developed by Gunnink and Prindle (1992). Working from real data, they defined three nonlinear curves that correlate energy (measured in MeV) versus efficiency on a log-log plot. The first curve covers energies below 90 keV and computes the "intrinsic" efficiency from the formula

$$e = \exp[-1.5 + 1.014\{\ln(0.9r)^2\}](1 - \Pr\{escape\})$$

where r is the radius of the detector (in cm), and the probability of electron escape is given by

$$\Pr\{escape\} = \exp[-21.16 - 8.0\ln E - 0.8257(\ln E)^2].$$

Another curve covers energies above 200 keV using a six-term polynomial of the form

$$e = \exp[a_1 + a_2\ln E + a_3(\ln E)^2 + a_4(\ln E)^3 + a_5(\ln E)^4 + a_6(\ln E)^5.$$

The coefficients are

$$a_1 = -4.317 + 0.96\ln(eff_{1.333})$$
$$a_2 = 0.06\left[1 - \left(\frac{2r}{h}\right)^2\right] - 1.13 - 0.0871(\ln V) + 0.0305(\ln V)^2$$
$$a_3 = 0.333 - 0.1154\ln V + 0.009427(\ln V)^2$$
$$a_4 = -0.1456 + 0.01592\ln V$$
$$a_5 = -0.015$$
$$a_6 = -0.003 + 0.0092\ln V - 0.00124(\ln V)^2$$

where $eff_{1.333}$ is the percent efficiency of the detector at 1.333 MeV (usually specified by the manufacturer), h is the height (or width) of the cylindrical detector (in cm), and $V = 0.8\pi r^2 h$ approximates the volume of the detector's crystal. The coefficient formulas above differ slightly from what was published by Gunnink (1992) because of errors contained in the article. Correct formulas were obtained directly from Gunnink's GRPANAL software (Gunnink et al. 1988).

Finally, a third curve is constructed to match the first two and cover energies from 90 keV to 200 keV. This is a simple quadratic interpolation given by

$$e = \exp[\ln e_{200} + \{8.092 + 7.55\ln E + 1.568(\ln E)^2\}(\ln e_{200} - \ln e_{90})],$$

where e_{200} and e_{90} are the intrinsic efficiencies at 200 keV and 90 keV, respectively, as calculated from the other two pieces of the efficiency curve.

The formulas above compute the intrinsic efficiency of a Ge detector at a particular energy based on the detector's relative efficiency at 1.333 MeV and the size of its crystal. The intrinsic efficiency is then converted to an *absolute* efficiency by multiplying by the factor 0.0012. This number is the absolute efficiency of a 3x3 NaI detector in converting the 1.333 MeV photons coming from a Cobalt-60 source located coaxially 25 cm away from the face of the detector. Thus, the percent efficiency rating $eff_{1.333}$ (defined relative to a 3x3 NaI detector) is converted back to absolute efficiency.

The efficiency of NaI detectors is calculated from data generated off-line by the EGS4 Monte Carlo simulation software developed at Stanford University. A table of efficiencies was generated for each specific crystal size, usually covering the range of energies from 40 keV to 10 MeV. Efficiencies at intermediate energies are calculated by QUEST using linear interpolation to a log-log plot.

For both Ge and NaI detectors it is customary to include absorption effects from other parts of the detector assembly in the efficiency factor. Using the $e^{-\mu d}$ loss factor described in the previous section, we account for the absorption from the metal casing of the detector and, in the case of Ge, from the "dead" layer of germanium that surrounds the active crystal.

Photoelectric Peak Shape

The electrons generated by the photoelectric effect are actually detected over a distribution of energies due to thermal motion of the crystal atoms. This continuous spectrum may be computed by mathematically convolving the photon spectrum distribution with a Gaussian pulse whose width reflects the resolution of the detector. A stream of *gamma ray* photons (emitted from atomic nuclei) has an extremely narrow spectrum; hence, convolution simply gives back the Gaussian response of the detector centered at the energy of the photon. A stream *of X-rays* (emitted from bound electrons) has a widened spectrum with a Lorentzian distribution; convolution gives a shaped Gaussian.

The Gaussian curve characterizing detector resolution is usually specified by its Full Width at Half Maximum (FWHM). A general Gaussian function centered at E_0 is given by

$$f(E) = \frac{A}{s\sqrt{2p}}\exp\left[-\frac{(E-E_0)^2}{2s^2}\right],$$

where σ is the standard deviation (in MeV) and A is the total area under the pulse (the number of photons converted by the photoelectric effect). The Gaussian function attains its maximum value at E_0, and the points where $f = \frac{1}{2} f(E_0)$ are easily seen to be

$$E_0 \pm \sqrt{-2s^2 \ln 0.5}\,.$$

Therefore,

$$FWHM = 2\sqrt{-2s^2 \ln 0.5} \quad \text{and} \quad s = FWHM / 2.3548\,.$$

The detector manufacturer usually indicates FWHM at one specific energy; however FWHM actually varies with photon energy. The variation is modeled by a simple two parameter formula:

$$FWHM = \sqrt{\exp[k_1 + k_2 \ln E_g]}\,,$$

where E_γ is photon energy (in keV), and k_1 and k_2 are parameters described by the equations below. In a NaI detector the user must supply the nominal width of the Gaussian at 661 keV (a strong emission line for Cesium-137). The value is specified as a percentage, roughly meaning the percent of a 661 keV spectrum covered by the Gaussian. Referring to this quantity as $W_{\%661}$, the parameters for a NaI detector are:

$$k_1 = -4.2674 + 1.998 \ln W_{\%661}$$
$$k_2 = 1.24 .$$

At low energies the parameterization is less accurate, so we mandate FWHM always be at least $\sqrt{20} = 4.47$ keV.

In a Ge detector the user must provide the nominal width at 1332.48 keV (an emission line for Cobalt-60). Unlike NaI detectors, it is given in units of keV. Calling this quantity W_{1333}, Ge parameters are:

$$k_1 = -7.1884 + 1.9999 \ln W_{1333}$$
$$k_2 = 0.999169 .$$

The FWHM in this case is mandated to be at least 1.0 keV.

A special correction factor is applied to the standard deviation of the Gaussian, reflecting the effects of representing the continuous energy spectrum as a sequence of finite width channels. In the GRPANAL work (Gunnink et al. 1988, vol. 1, pp. 30-31) this is referred to as "Sheppard's" correction. As in SYNTH, we use the formula:

$$FWHM_{corrected} = (chan\_width)\sqrt{\left(\frac{FWHM}{chan\_width}\right)^2 + 0.46} .$$

Typically, channel width of a detector is 1 keV, and the correction is not significant for larger Gaussian widths (for example, $FWHM_{corrected}$ = 1.208 when *FWHM = 1 keV,* but $FWHM_{corrected}$ = 10.023 when *FWHM* = 10 keV). Note that the correction becomes quite large as channel width approaches infinity, the opposite of what might be expected. It is likely that this correction formula is appropriate only for the SYNTH fixed channel width of 1 keV.

Summarizing, to calculate the detector response due to photoelectric effects induced by a gamma ray:

- compute A = (number of source photons per second) × (absolute efficiency at E_γ)
- compute σ from the *FWHM* of the detector at E_γ
- the response is $f_p(E) = \frac{A}{s\sqrt{2p}} \exp\left[-\frac{(E-E_g)^2}{2s^2}\right]$.

<u>Photoelectric Peak Shape for X-Rays</u>
Photons emitted by an X-ray decay process have a Lorentzian spectral distribution that can be written using the Breit-Wigner formula:

$$h(E) = \frac{\Gamma/2p}{(E-E_g)^2 + (\Gamma/2)^2} .$$

The distribution is a symmetric pulse shape with total area equal to one. Its width is specified by Γ, the full width at half maximum for this function. Following SYNTH and GRPANAL (Gunnink et al. 1988, V3, p. 128), we choose it by the rule:

$$\text{if } E_g \le 0.028 \text{ MeV}$$
$$\text{then } \Gamma = (0.4E_g - 0.0004)/1000$$
$$\text{else } \Gamma = (1.372E_g - 0.02762)/1000 .$$

The widths are fairly small; for instance, an X-ray photon at 90 keV has a full width at half maximum of 0.096 keV.

The detector response is given by the convolution of its Gaussian impulse response (characterized as above by σ = *FWHM*/2.3548) with the distribution $\eta(E)$; that is,

$$f_{pX}(E) = \int_{-\infty}^{\infty} f_p(t)h(E-t)dt \ .$$

The convolution result is known as the Voigt profile, and is approximated numerically using the formulas from work on fitting Lorentzian peaks (Gunnink 1977) and GRPANAL (Gunnink et al. 1988, V3, p. 18 and p. 24).

Photoelectric Peak Tail

In addition to the Gaussian-like pulse f_p or f_{pX} centered on the photon's energy, Gunnink (1972) adds a "tailing term" on the low energy side of E_γ. It's a response given by

$$f_t(E) = \frac{A}{s\sqrt{2p}} a\exp[b(E-E_g)]\left\{1-\exp\left[-c\frac{(E-E_g)^2}{2s^2}\right]\right\}$$

if $E \le E_\gamma$ (otherwise, $f_t(E) = 0$). The special parameters in this expression are

$a = \exp[-2.9 + 0.44E_g]$	("amplitude"),
$b = 1.62*1000$	("slope"), and
$c = 0.4$	("fold-over constants").

The term involving b causes the tailing response to drop off exponentially as E becomes substantially less than E_γ. We ignore the tail completely when $\exp[b(E - E_\gamma)]$ is smaller than machine precision; i.e., when $E_\gamma - E > 0.025$ MeV.

Compton Effect

This occurs when a photon collides with a nearly free electron in the crystal, imparting a fraction of its energy to the electron, which can then be detected. The photon is actually annihilated and a new "scattered" photon of lower energy is reradiated in a different direction. QUEST does not model further interactions with the scattered photons.

The fraction of photon energy acquired by the electron depends on the angle of the collision between the two particles. The electron picks up a maximum amount of energy in a head-on collision. This is known as the Compton energy, and has the value (see, for instance, Weidner et al. 1973, p. 123)

$$E_c = E_g \frac{2E_g/m_e}{1+2E_g/m_e} .$$

The quantity m_e = 0.511006 MeV is the relativistic energy equivalent of an electron at rest. Photons that collide less directly with an electron deliver less energy to it; thus, electrons from the Compton effect are detected over a range of energies between zero and E_c. The response in this is modeled from the distribution in Kopecky et al. (1967). They define

$$y\ (E) = \frac{m_e}{E_g}\left\{2 + \left(\frac{E}{E_g - E}\right)^2\left[\left(\frac{m_e}{E_g}\right)^2 + \frac{E_g - E}{E_g} - 2\frac{m_e}{E_g}\frac{E_g - E}{E}\right]\right\}$$

to be the relative response from a single Compton scattering process, neglecting scattered photons. This response is normalized to equal 1 at the Compton energy and then scaled using the peak-to-Compton ratio *PCR* (formally, *PCR* is $f_p(E_\gamma)$ divided by the Compton response at E_c). Thus, the scaled Compton response is

$$f_c(E) = \frac{A/s\sqrt{2p}}{PCR}\frac{y\ (E)}{y\ (E_c)}, \quad \text{where} \quad y\ (E_c) = 2\frac{m_e + E_g}{E_g}.$$

The peak-to-Compton ratio is calculated from heuristic formulas that depend on detector characteristics. For Ge detectors the ratio is (EG&G Ortec 1991):

$$PCR = PCR_{1333}\exp\left[-0.319\left(\ln\frac{E_g}{1.333}\right) - 0.081\left(\ln\frac{E_g}{1.333}\right)^2 - 0.062E_g^{-0.0011}\right],$$

where

$$PCR_{1333} = 34.75 + 1.068eff_{1333} - 0.00496eff_{1333}^2$$

is a specific estimate of the peak-to-Compton ratio at 1.333 MeV.

For NaI detectors, a simpler model from SYNTH is used:

$$PCR = \exp[1.6 - 0.67(\ln E_g)].$$

Pair Production and Escape Peaks

If a photon has energy greater than $2m_e$ = 1.02201 MeV, then it may create an electron-positron pair of particles in the neighborhood of a heavy atom. The mass of the newly created particles requires exactly $2m_e$ units of energy from the photon. If E_γ exceeds this value, then the remaining energy is distributed between the electron and positron as kinetic energy, which can be detected. Thus, we observe a response at $E_\gamma - 2m_e$, referred to in SYNTH as a "double escape peak". Because the created particles are subject to thermal motion in the detector, the response is the Gaussian pulse $f_p(E)$ with width *FWHM* that was calculated for the photoelectric effect.

A similar pair production phenomenon involving positrons generates a "single escape peak" response at $E_\gamma - m_e$ (the physics still requires that $E_\gamma \geq 2m_e$). Again, the peak is a Gaussian pulse.

The fraction of photons that create escape peaks is computed using heuristic formulas from Gunnink (1972) and SYNTH (Hensley et al. 1994). The ratio between the maximum height of the two Gaussians is specified as

$$\frac{f_{sngl}(E_g - m_e)}{f_p(E_g)} = \exp[-5.2822 + 6.2381\{\ln(E_g - m_e)\} - 2.2886\{\ln(E_g - m_e)\}^2]$$

$$\text{and} \quad \frac{f_{dbl}(E_g - m_e)}{f_p(E_g)} = \exp[-3.0 + 2.1\{\ln(E_g - 2m_e)\}].$$

Multiple Compton Region

The region between E_γ and E_c contains a low-level response described as a "multiple Compton" effect in SYNTH. It is modeled by the empirically determined formula

$$f_{mc}(E) = \frac{A/s\sqrt{2p}}{PCR}\frac{1}{1.2}\exp\left[-0.7p\,\frac{E-E_c}{E_g-E_c} + 0.3\sin(2p\,\frac{E-E_c}{E_g-E_c})\right].$$

This function does not match the Compton response at $E = E_c$, producing an unappealing discontinuity in the response spectrum. Hensley computes the right side of the Gaussian impulse response at the Compton energy edge, and chooses the multiple Compton response to be the maximum of this function or f_{mc}. Then discontinuities are further "smoothed" by applying a simple three-point averaging filter from E_c – 10 keV to E_c + 10 keV.

3.2.6 Modeling Detector Signal Processing

The processing in this section converts the continuous energy spectrum generated from the previous section into a ChannelSpectrum object. The processing models energy channelization, signal amplification, and time integration. Some of the issues have been introduced in earlier sections.

Channelization

A detector accumulates photon counts in a number of channels that cover a given energy range. The user specifies the lowest energy of interest, the nominal width of each channel (also called the "gain" of the detector), and the total number of channels. For example, 1000 channels of width 2 keV starting at 50 keV covers the energy spectrum from 50 keV to 2050 keV. In this example "channel 1" counts photons with energies over the interval [50 keV, 52 keV), "channel 2" over [52 keV, 54 keV), etc.

The computations of the section above started with source LineSpectrum objects at a particular time instant and resulted in a single continuous energy distribution. The counts recorded in a given detector channel should be the integral of this continuous distribution over the energy range covered by the channel. We approximate the integration numerically using the trapezoid method, which makes a linear interpolation between selected sample points of the distribution. Sample points are chosen to be uniformly separated by 1 keV or the width of the channel, whichever is smaller.

Amplification

Signal amplification within the detector may cause a nonlinear skewing of the energy spectrum that is actually accumulated by a channel. Following SYNTH, we model this distortion by a second-order term specified by the user. If we let w be the nominal channel width and E_0 the lowest energy recorded by the detector, then an ideal detector would count photons in channel n over the range of energies given by

$$lowest\_channel\_energy = E_0 + (n-1)w$$

$$highest\_channel\_energy = E_0 + nw\,.$$

In our model we compute channel ranges with a second-order distortion term characterized by the constant q. Distorted ranges are

$$lowest\_channel\_energy = \left(\frac{E_0}{w} + n - 1\right)w + \left(\frac{E_0}{w} + n - 1\right)^2 q$$

$$highest\_channel\_energy = \left(\frac{E_0}{w} + n\right)w + \left(\frac{E_0}{w} + n\right)^2 q\,.$$

If q is negative, then the energy spectrum is shifted to the left with respect to nominal channel definitions; i.e., channels contain counts from photons of higher energy than expected. If the distortion factor is positive, energies shift in the other direction.

Time Integration

A detector accumulates photon counts over a sampling period, typically measured in seconds. The ChannelSpectrum object computed at one time instant is really just one sample point with respect to an integration over time. The numerical integration scheme is described in detail in the previous section entitled "Accounting for Source and Detector Motion".

3.3 MATERIAL DATABASE

The material database contains descriptions of materials that may be encountered in the QUEST virtual environment. This description includes the density, thickness, and composition by weight of the material—information needed by the physics model to calculate the change in photon or elementary particle spectra as these objects pass through materials.

Error Estimate

There are several possible sources of discrepancy between the data found in the material database and the actual densities, thicknesses, and compositions found in a particular, real world, building structure. Assuming that the number and types of building materials found in a structure and in a data base entry are identical, the most significant such discrepancies are due to allowed tolerances in thickness of building materials and local variations in compositions of these materials. Tolerances on the order of a few percent are typical. For example, a brick that is nominally $3^{5}/_{8}$" thick may actually be anywhere between $3^{4}/_{8}$" and $3^{6}/_{8}$" thick, a 3% variation (International Conference of Building Officials 1994). Chemical compositions of materials can vary widely depending on the nature of locally available materials. Again using brick as an example, the percent by weight of silicon in bricks varies from 26% to 35%, i.e., by approximately one third, while other elemental compositions vary even more widely (Brick Institute of America 1996). One can expect such variations in material compositions to introduce an additional uncertainty of a few to ten percent in scattering amplitudes.

The inherent uncertainty of about 10% discussed in the previous paragraph sets the scale for needed precision in the density and elemental composition of building materials. Densities will be rounded to the nearest 0.1 gm/cm^3. Elemental compositions will be rounded to the nearest 0.1 %. Correspondingly, any structural components that affect the final density and elemental compositions by less than these amounts need not be considered.

Calculation of a Materials Data Base Entry

To calculate a material database entry for a new structure one needs to express that structure as a sum of pieces whose composition and density are already known or can be found easily in one of the Appendices C or D. As an example, consider a residential interior wall. A typical residential interior wall is made up of the following components (Packard 1981):

2 x ½"	gypsum boards,
2" x 4"	pine studs every 16",
1¾"	stainless steel nails every 8" vertically,
3½"	cavity (air).

It should be noted that after 1982 studs may be placed 24" apart instead of 16" apart (from conversation with local builder and the 1994 Uniform Building Code (International Conference of Building Officials 1994)). To complete the materials data base entry for a standard residential interior wall, three items must be computed: 1) total thickness, 2) density, and 3) elemental composition.

3.3.1 TOTAL THICKNESS

The total thickness is computed by adding together the thicknesses of the appropriate components. In this case,

$$\begin{aligned} \text{Total Thickness} &= 2 \times (\text{Thickness of 1 gypsum board}) + (\text{Thickness of } 2 \times 4) \\ &= 2 \times (½") + (3½") \\ \text{Total Thickness} &= 4½". \end{aligned}$$

3.3.2 DENSITY

The total density may be computed with the following formula:

$$\text{Density} = \frac{1}{\text{Total Thickness}} \sum_{\text{Components}} (\text{mass per unit area for each component})$$

In this example, the gypsum boards have a density of 2.3 gm/cm^3 and are 1" (or 2.54 cm) in total thickness. Their mass per unit area (or surface density) is then,

$$\text{surface density} = (2.3 \text{ gm/cm}^3)(2.54 \text{ cm}) = 5.84 \text{ gm/cm}^2.$$

The density of pine is 0.55 gm/cm^3, the studs are 3½" thick, $1^5/_8$" in width and placed 16" apart. Thus,

$$\text{effective surface density of pine} = (1\tfrac{5}{8}/16)(3\tfrac{1}{2}")(2.54 \text{ cm/in})0.55 \text{ gm/cm}^3 = 0.50 \text{ gm/cm}^2.$$

The nails each weigh 1.8 gm (Packard 1981) with 2 nails for every 8" x 16" (826 cm^2). The effective surface density of the nails is then,

$$(3.6 \text{ gm})/(826 \text{ cm}^2) = 0.004 \text{ gm/cm}^2.$$

The effective surface density of the nails is 0.1% that of the gypsum boards and, based on the discussion in Section 1, can be ignored. For the air occupying the wall cavity, we have,

$$\text{effective surface density} = (14\tfrac{3}{8}/16)(3\tfrac{1}{2}")(2.54 \text{ cm/in})0.0013 \text{ gm/cm}^3 = 0.01 \text{ gm/cm}^2,$$

which can also be neglected.

The effective density of the wall is thus,

$$\text{Density} = \frac{1}{(4.5")(2.54 \text{ gm/in})}(5.84 \text{ gm/cm}^2 + 0.50 \text{ gm/cm}^2) = 0.56 \text{ gm/cm}^2.$$

3.3.3 ELEMENTAL COMPOSITION

The elemental composition of a composite structure is given by a weighted average of the compositions of the constituent pieces. The weight of each constituent is given by the ratio of the mass of that constituent to the entire mass. For the current example, this becomes,

$$\left(\tfrac{5.84}{6.34}\right)\begin{pmatrix} O & 55.8\% \\ Ca & 23.3 \\ S & 18.6 \\ C & 0 \\ H & 2.3 \\ N & 0 \end{pmatrix} + \left(\tfrac{0.5}{6.34}\right)\begin{pmatrix} O & 43\% \\ Ca & 0 \\ S & 0 \\ C & 50 \\ H & 6 \\ N & 1 \end{pmatrix} = \begin{pmatrix} O & 54.7\% \\ Ca & 21.5 \\ S & 17.2 \\ C & 3.9 \\ H & 2.5 \\ N & 0 \end{pmatrix},$$

with the first term corresponding to the gypsum boards and the second to the pine studs. The elemental composition of gypsum was derived from its chemical formula, $CaSO_4 \bullet 2H_2O$, as follows:

Element	Weight (gm/mole)	Percent by weight ([previous column]/[total weight])
O	96.00 (6 x 16.00)	55.8
Ca	40.08	23.3
S	32.06	18.6
H	4.03 (4 x 1.008)	2.3

3.4 BACKGROUND RADIATION

In general, there are other sources of radiation in the environment besides the source objects that QUEST allows the user to place. These other sources are lumped together and considered "background". The character of the background radiation depends on the sources that cause it. There is a certain distribution of radioactive elements in the earth's soil worldwide. Localized sources may include geological formations and man-made items such as glazed bathroom tiles and rock gardens. In addition, radiation reaches the earth from space, predominantly from the sun.

Within QUEST, background is specified as a *source* or a *spectrum* (or both). A background source is a regular QUEST source, defined like other QUEST sources. Instead of being placed on a particular path, however, the background source receives special treatment to make it appear ubiquitous. Any spectrum file in standard IEEE or ADCAM format may also be used as a background spectrum. A background spectrum is simply added to the spectrum developed by the normal operation of the detector. Ideally, the spectrum should have the same zero, gain and number of channels as the detector; if it does not, it is interpolated to fit the detector.

Approach

In the first release of QUEST, localized background sources are ignored. The user is allowed to add to the observable radiation a background spectrum that does not change with ground location. A default background spectrum intended to approximate the radiation emitted by soil is provided, and the user is allowed to substitute his/her own spectrum in place of it, or use no background spectrum at all.

Based on the analysis in Miller and Shebell (1993) and Helfer and Miller (1988) and knowledge of the simulation of the detectors, the following assumptions and approximations are made:

1. The background radiation provided by the soil can be treated as a homogeneous quantity that varies only with distance above the ground.
2. The variation caused by the angular orientation of the detector can be accounted for by multiplying the background spectrum by a factor that depends on the detector's orientation with respect to horizontal.
3. The mass absorption coefficient of air is less than 1.0×10^{-4} cm^{-1} at all energies of interest. This means that attenuation by air is no more than 10% at altitudes up to 10 meters.
4. Inside a building, walls attenuate the background radiation much more strongly than the air.
5. The contribution of cosmic radiation to the background can be ignored below an altitude of 3 kilometers.
6. The spectrum of the background provided by the soil can be approximated sufficiently closely by including its four main radioactive components and their naturally occurring daughter products in their worldwide average distributions. The four components are ^{40}K, ^{238}U, ^{235}U and ^{232}Th.
7. The gamma rays from the sources mentioned in (6) effect detectors over a wide range of energies. The computation of this response is significant and should only be done once during a simulation.

Assuming that detectors will not be suspended high in the air, (4) and (3) imply that the attenuation of the background radiation by air can be neglected. Assuming that no searches of airborne aircraft will be simulated, (5) implies that the contribution of cosmic radiation to the background can be ignored.

In order to accommodate (6) and (7), the default background spectrum will be provided in the form of a Source specification. The default background source will be a point source consisting of the components mentioned in (6) with strengths corresponding to their worldwide average distributions. The user may substitute any other Source for the default.

The user may also specify a uniform background spectrum that is added to the ambient spectrum at all points. The user may specify both forms of background (Source and uniform), either form, or neither. The spectrum for a uniform background may be provided in either of two standard file formats, IEEE (Institute of Electrical and

Electronics Engineers 1993) or ADCAM. When a uniform background spectrum is specified, the user accepts responsibility for ensuring its compatibility with specific detectors.

Software Considerations

A Background object will be created by the GUI and made part of the Scenario object controlling the simulation run. The object describes the background radiation in the simulated environment. If the user specifies a background Source object, the Background object will record its name. If the user specifies a uniform background spectrum, the Background object in the GUI will read the indicated file to make sure it is valid but will not retain its contents. A Background object residing in the TP will read the file and retain its contents for use in the simulation.

If a uniform background spectrum is used, it will be kept in the Background object in the form of a ChannelSpectrum object. In order to be usable by a particular detector, the channel spectrum's characteristics (zero, gain, number of channels) must match those of the detector. To accomplish this, each detector will create its own Background object. If the characteristics of the spectrum in the specified IEEE or ADCAM file do not match those of the detector, the Background object will adjust them as best it can. It will perform this conversion as soon as it learns the detector's characteristics, namely the first time the detector asks it to add the background contribution to one of the detector's channel spectra. If any fundamental incompatibility is discovered between the characteristics of the detector and those of the uniform background spectrum, a warning message will be given, but the background spectrum will be used anyway.

Operations

A Background object may be created with the following operations:

```
Background (void);
Background (const Background&);
```

The GUI uses the first form to create the Background object that is part of the Scenario. A Detector uses the second form to create its own Background object.

The operations

```
RWBoolean source (RWCString name);
RWCString source (void);
```

are used to set and fetch the name of a Source object which is to be used to supply background radiation. The GUI uses the first operation to set the name, and the PBE (the Detector which owns the Background object) uses the second operation to fetch the name. Related operations are

```
RWBoolean backgroundFromSource (RWBoolean enable);
RWBoolean backgroundFromSource (void);
```

The first operation causes the source named in the *source* operation to be used to provide background radiation if its argument is TRUE and causes it not to be used if its argument is FALSE. Initially, the background source is disabled, so a named source will not be used until it is enabled with a *backgroundFromSource* operation with TRUE argument. The second operation returns TRUE if the background source is enabled and FALSE if it is not.

The operations

```
RWBoolean spectrumNameAndType (RWCString name, RWCString type);
RWCString spectrumName (void);
RWCString spectrumType (void);
```

are used to set and fetch the name and type of the file containing the spectrum to be used to supply uniform background radiation. The *spectrumNameAndType* operation sets the name and type. The types recognized are

"IEEE" and "ADCAM". The operation returns a false value if the name is null or the type is not one of the recognized types. The other two operations return the name of the spectrum file and its type, respectively. The operations

```
RWBoolean spectrumMultiplier (float multiplier);
float spectrumMultiplier (void);
```

set and fetch a factor by which the uniform background spectrum is multiplied. The first operation always returns TRUE.

Like the background source, the uniform background spectrum may be enabled and disabled. The operations

```
RWBoolean backgroundFromSpectrum (RWBoolean enable);
RWBoolean backgroundFromSpectrum (void);
```

are used for the purpose. The first operation causes the spectrum file source named in the *spectrum* operation to be used to provide background radiation if its argument is TRUE and causes it not to be used if its argument is FALSE. The operation always returns a TRUE value. Initially, the background spectrum file is disabled, so a named spectrum file will not be used until it is enabled with a *backgroundFromSpectrum* operation with TRUE argument. The second operation returns TRUE if the background spectrum file is enabled and FALSE if it is not.

The operation

```
RWBoolean complete (void);
```

returns TRUE if the Background object has a usable set of parameters and FALSE otherwise. To have a usable set of parameters, it must have a complete set of source parameters and a complete set of spectrum parameters. The source parameters are considered complete if the source is disabled, or if it is enabled and the source name is non-null. The spectrum parameters are considered complete if the spectrum is disabled, or if the spectrum is enabled, the spectrum name is non-null and the spectrum file has been validated. Validation of a spectrum file consists of making sure it exists and is readable as a file of the type given by the spectrum type parameter. Validation is done only if necessary, as soon as possible and not more than once per spectrum name. It is deemed necessary as soon as the spectrum is enabled and there is a non-null spectrum name. Thus it may be done when the spectrum is enabled with the *backgroundFromSpectrum* operation or when the spectrum name and type are specified with the *SpectrumNameAndType* operation. If it fails when it is done, it causes the operation that provoked it to return a false value. It is not done again unless another spectrum name is specified with the *SpectrumNameAndType* operation.

The operation

```
void addBackground (ChannelSpectrum&);
```

is used by the Detector which owns the Background object to add in the contribution of a uniform background spectrum. If the background spectrum is not enabled, the operation has no effect. It is up to the Detector object to develop and add in the effects of the background source, if one is enabled.

3.5 COMPONENT DATABASE

The purpose of the Component Database (CDB) is the representation of physical structures (e.g., buildings and their subcomponents) in a QUEST virtual environment. Two particular requirements are placed on the CDB: it must be possible for the user to select subparts of the structures for manipulation (the various types of manipulation are described below); and it must be possible to compute the interaction of gamma rays and streams of elementary particles within the structures.

Data Structures

Every physical structure can be thought of as made up of components that stand in a hierarchical relationship to one another. A structure is a component that is made of subparts that are components, and so forth. During a quest run, all the structures in the QUEST environment are represented by a single component tree, which expresses the structural relationships, and one or more binary space partitioning (BSP) trees, which support the computation of the structure's interactions with streams of photons or other particles.

The component tree is implemented as an object of C++ class Component. The Component object provides all the services associated with the CDB and can be used by itself, which is useful in testing. In the QUEST system, however, it is used through the interprocess communication mechanism. Two associated C++ classes are required, ComponentServer, which is derived from Component, and ComponentClient. The Component object lies in the QUEST Physics Back End (PBE) process, along with a ComponentServer object. There may be any number of objects of type ComponentClient. Each program which requests services of the CDB has a ComponentClient object through which it makes those requests. The ComponentClient object relays a request to the ComponentServer object in the PBE, which calls upon the Component object to perform it. After the operation is done, the ComponentServer object returns a response to the ComponentClient object. Interprocess communication and client-server interaction are described in Section 3.6. Each BSP tree is implemented as an object of C++ class BSP. Objects of class BSP are used only by the Component object.

The Component Tree

The component tree has several different types of nodes. Some of the nodes contain the structure's geometry; other nodes represent attributes of the structure, such as its color or composite material.

The component tree can be described by the following expression in Backus Normal Form (BNF) (Aho and Ullman 1972):

```
Component = {
                Geometry
            |   startSubcomponent Component
                endSubcomponent
            |   Attribute
            |   Transform
            |   label
            |   reference
          }

Geometry = ( vector { vector } | polygon { polygon } )

Transform = ( scaling | rotation | translation )

Attribute = ( PhysicalAttribute | groupName )

PhysicalAttribute = ( materialName | paintColor | paintTransparency | thickness )
```

The nonterminal (further defined) elements start with a capital letter and the terminal elements with a lower-case letter. The hierarchical nature of the structure is reflected in the recursive definition of Component. As the definition shows, every branching node in the tree together with its descendants can be thought of as a component.

Structural Component Groupings

The group names mentioned in the BNF definition of Component are Structural Component Grouping (SCG) names. SCGs are a hierarchy of names with which attributes are associated (see Section 3.5.2). The SCG hierarchy and the attributes associated with the groupings are recorded in the SCG file. The user may provide a custom SCG file for each run, and may modify the SCGs in setting up the run (operations for doing so are given below). A default SCG file is provided; if not even the default file is used, a single group named "DEFAULT" is always defined. The use of SCGs in connection with structures is explained below.

The QUEST Structure File

The component tree that is constructed for a QUEST run is read from one or more QUEST Structure Files (QSF). A QSF represents a component; such a component is called an *external* component. Every structure used in QUEST is initially defined in an AutoCAD DXF file (AutoDesk 1990). The process of converting a DXF file to a QSF file is called *installing* the file. During this conversion, DXF layer names are interpreted as group names and DXF layer colors are interpreted as paint colors. Any layer names not present in the SCG file are treated as first-level groups (descendants of the default group) and given the default group's attributes.

The format of a QSF file is a subset of the Virtual Reality Modeling Language (VRML) format (Bell et al. 1995). VRML is an ASCII file format patterned after the Silicon Graphics Open Inventor file format (Wernecke 1994). For the purpose of storing it in a file, the hierarchy of the component tree must be flattened out into a one-dimensional form. Examples of such flattened hierarchies are the display lists of many graphics controllers, the DXF and Open Inventor file formats and VRML. NFF (Sense 8 Corporation 1995), used by World ToolKit, is not such a format, since it has only three levels of hierarchy (entire file, NFF group and NFF object). It is desirable to use a standard format to make it possible to use our structures with other packages. It might be possible to extend NFF to have the required capability, but then it would no longer be standard. VRML already has all the needed features; besides the usual geometry and appearance features, it allows informational nodes and self-defining nodes, which are useful for Structural Component Groupings (and for any unpredictable future requirements). DXF is overly complex and does not have self-defining nodes. Open Inventor (OI) does not have informational or self-defining nodes, but allows the user to define his own types as descendants of OI types; however, the user must also supply the code to read and write the new types and integrate the code with the OI file reading and writing programs. VRML appears to provide the simplest choice that is both adequate and standard.

3.5.1 BINARY SPACE PARTITION TREES

Each Binary Space Partition (BSP) tree represents the same geometry as some subcomponent of the component tree but in a fashion optimized for rapid spatial searching. A BSP tree may be created for any subcomponent of the tree. It is created by explicit user request (see below). If the subcomponent for which a BSP tree is created has subcomponents that already have BSP trees, the polygons of those components are not included in the new BSP tree. When a spatial search is done, all existing BSP trees are searched.

A BSP tree is built one polygon at a time. Before being placed in the BSP tree, the polygon is "reconditioned" to remove irregularities. Any figure that has less than three vertices or is nonplanar is thrown out. Successive collinear vertices are removed. If the sides of the polygon intersect, the figure is broken up into polygons with non-intersecting sides. Finally, if a polygon is concave, it is converted into triangles, which are perforce convex. The triangularization code used in this version of QUEST originated at Evans and Sutherland and was obtained from the Graphics Research and Analysis Facility at NASA's Johnson Space Center.

BSP trees are heavy users of storage. In order to cut down fragmentation, vectors (vertices) are allocated in "chunks" in an object of the Pool class. Each BSP tree has its own vector pool, which is created with it, grows with it and is destroyed with it.

Creating a Structure

A new component tree is created by the sequence of operations

```
clear  {  Component  }
```

where *clear* is defined as

```
void clear (void);
```

and Component is one of the following operations:

```
void startVectors (void);
typedef struct Vector {
    float x [4];
} Vector;
void vector (int Nvectors, Vector *v);
void vector (Vector v);
int polygon (int Nvertices, int *vectorIndex);
int polygon (int Nvertices, int *vectorIndex, Vector normal);
void group (char *groupName);
void material (char *matName);
     const int RGBsize = 3;
void paintColor (float color [RGBsize]);
     void paintTransparency (float trans);
void thickness (float thick);
void label (char *text);
     int startSubcomponent (void);
     int startSubcomponent (char *name);
void endSubcomponent (void);
int reference (char *name);
void scale (Vector scale);
void translate (Vector translation);
void rotate (Vector axis, float rotation);
typedef struct Matrix4x4 {
      float elements [4][4];
} Matrix4x4;
void transform (Matrix4x4 transformation);
```

Note that there is one operation for every terminal in the BNF definition of Component. Each of the above operation (except *clear*) creates a node of the component tree. Each node has a unique identifier known as its *position number*. Each of the above operations above operations places a node at the current position of the component tree and defines that node as the new current position. Certain operation place nodes at given positions in the component tree, specified by a position number:

```
int startSubcomponent (int positionNumber);
int startSubcomponent (int positionNumber, char *name);
void scale (int positionNumber, Vector scale);
void translate (int positionNumber, Vector translation);
void rotate (int positionNumber, Vector axis, float rotation);
void transform (int positionNumber, Matrix4x4 transformation);
```

If the node with the given position number is a subcomponent node, the new node is made its descendant; otherwise, the new node is made the descendant of the parent of the node with the given position number.

Unless the component tree is empty, a *current* node is always defined. The position number of the current node is returned by the operation

```
int current (void);
```

(If the component tree is empty, it returns zero.) The current node may be changed by means of the operations

```
int makeCurrent (int positionNumber);
int makeRootCurrent (void);
```

If the given position number describes an existing node, the *makeCurrent* operation makes that node current, otherwise it leaves the current node unchanged; in any case, it returns the position number of the current node at the conclusion of the operation. The *makeRootCurrent* operation makes the root node of the component tree the current node and returns its position number (or zero if the component tree is empty).

As the operations are performed, the component tree is constructed. A *state value* is associated with every location in the component tree. The state value contains

- a vector sequence
- a geometric transformation
- a complete set of attributes, including
  - a material
  - a paintColor
  - a paintTransparency
  - a thickness
  - a group name
- a label

Initially, the vector sequence is empty, the transformation is the identity transformation, the material is the default material, the paintColor, paintTransparency and thickness are those associated with the default material, and the group is the default (root) group.

The state value is *inherited*. That is, it continues to be in effect at subsequently defined nodes of the tree unless it is explicitly replaced (overridden), with the exception that any state changes made within a subcomponent do not affect anything outside that subcomponent (as if the state value were pushed on entry to a subcomponent and popped on exit from it). A vector, group, label, color, transparency or thickness node merely overrides the previous value of the respective type in the state; a transformation node is composed with the transformation value in the state by pre-multiplication.

Once a component tree is created, it may be stored to disk with the operation

```
int store (char *filename, char *path, char *fullname, char *description);
```

A description of each of the above operations follows.

The *c*lear operation creates an empty component tree.

The *startVector* operation defines a new vector sequence. Subsequent *vector* operations append vectors to the vector sequence. The *startVector* operation inserts a node at the current position of the component tree to contain the vectors.

The *polygon* operation includes a polygon node at the current position in the component tree. The *vectorIndex* argument in the *polygon* operation contains indices referring to the current vector sequence. There are two types of polygons. The first type is considered to have a thickness (of a certain material; see below) and the second type is considered to connote entry into or exit from a material. The second type requires a normal, which is given by the *normal* argument of the *polygon* operation. By convention, the normal points outward, out of the material.

The integer value returned by *polygon* is TRUE if the polygon is valid; an invalid polygon is not inserted into the component tree, and a zero value is returned. Although there are many ways a polygon can be invalid, the only validity check that is done is to verify that the vector indices actually lie within the current vector sequence.

The *group* operation includes a group node containing the *groupName* argument at the current position of the component tree. The *groupName* string is interpreted as the name of a structural component grouping (see Section 3.5.2).

The *material* operation includes a material node containing the *matName* argument at the current position in the component tree. The *matName* string is interpreted as the name of a material in the Material Database (see Section 3.3).

The *paintColor* operation inserts a color node at the current position of the component tree. The argument of *paintColor* gives RGB color components as values in the range [0,1], where 0 means none of the primary color is present and 1 means the maximum amount of the primary color is present.

The *paintTransparency* operation inserts a transparency node at the current position of the component tree. The argument gives the transparency as a value in the range [0,1], where 1 means completely transparent and 0 means completely opaque.

The *thickness* operation inserts a thickness node at the current position of the component tree. The argument gives the thickness of the material in inches.

The *label* operation places an arbitrary text string at the current position in the tree.

The *startSubcomponent* operation appends a branching node to the tree structure. All nodes appended until the next *endSubcomponent* are descendants of the branching node. The subcomponent may be given a name that can be used in the *reference* operation and the *findSubcomponent* operation (see below). The *startSubcomponent* operation returns the position number of the new branching node.

The *translate, rotate* and *scale* operations specify the named transformations in an obvious fashion (the rotation is counter-clockwise and given in radians). The operations are composed (premultiplied) with the existing transformation in the current state value to form the transformation value active at the current position in the component tree. If there is already a transformation node in the current subcomponent, it is modified by this operation; if there is not, a new transformation node is inserted.

The *transform* operation gives a four by four transformation matrix that may express a combination of translation, rotation and scaling. It is postmultiplied with the existing transformation in the current state value to form the transformation value active at the current position in the component tree. If there is already a transformation node in the current subcomponent, it is modified by this operation; if there is not, a new transformation node is inserted.

The forms of the *translate, rotate, scale* and *transform* operations which specify position number also return the position number of the new or modified transformation node. The difference between the premultiplication of the *translate, rotate* and *scale* operations and the postmultiplication of the *transform* operation is significant.

The *store* operation stores the component as a QSF file under the file name *filename*.qsf. The file name may be a maximum of eight characters in length. The file is stored in the directory specified by the *path* argument. If *path* is null, the file is stored in the current directory. By convention, the component has a full name that may be up to 256 characters; this name is supplied by the *fullname* argument. Also by convention, the file contains a description that may be of any length; this description is supplied by the *description* argument, and it may contain newline characters. If the file already contains a full name (as it would if read with the *load* operation; see below), the given full name replaces it unless it is null, in which case the old full name remains; a similar remark applies to the description. The operation returns TRUE if the file was stored successfully.

A subcomponent node is the root of a structure that has the same form as the component tree. Such a structure may be considered to be an *internal component.* An internal component may be given a name in the *startSubcomponent* operation. As mentioned above, a component stored as a QSF file is called an *external component.* The *reference* operation includes a reference to an internal or external component, which it specifies by name. The referenced component must already exist as an external component (i.e., as a QSF file) or as an internal component already defined and named in the component tree. Referring to the component has the same effect as adding a subcomponent that contains the referenced component (it is the same as if the component were defined "inline"). If the same name is used for more than one internal component, the *reference* operation is taken to refer to the most recently defined one. By using the name of an external component, a QSF file's contents can be included at the current position in the component tree. The *reference* operation returns the position number of the new reference node or zero if the referenced component is not defined. The component data structure must be acyclic, that is, a component must not refer to itself or to a component which refers directly or indirectly to the first component. It may not always be possible for the software to verify that this restriction is obeyed, but all bets are off if it is violated.

Loading a Structure

An existing external component may be loaded with the operation

```
int load (char *filename, char *path, int external);
int load (int positionNumber, char *filename, char *path, int external);
```

If a load is done immediately after a *clear*, the component is made the root component; the component tree in memory is then identical to the component tree which existed at the time the file was written to disk. In general, the first form of the operation creates a new subcomponent node at the current position in the component tree and makes that node the root of the loaded component; the second form put the subcomponent node at the given position. The *filename* argument gives a file name which may be a maximum of eight characters to which the suffix "`.qsf`" is appended before reading the file. The *path* argument gives the directory in which to look for the file and may be null, in which case the current directory is used. If the *external* argument is FALSE, the contents of the QSF file appears in the component tree; if the *external* argument is TRUE, the component tree contains only a symbolic reference to the QSF file. The second option (*external* argument TRUE) is not used in the QUEST system. The *load* operation returns the position number of the root node of the loaded component if the file was loaded successfully and zero otherwise. Another form of the load operation

```
int loadDXF (char *filename, char *path);
int loadDXF (int positionNumber, char *filename, char *path);
```

is used to load a DXF file and convert it to an internal component. To be susceptible to such a treatment, the DXF file must satisfy certain criteria; see Structure Creation in Appendix B.

Reading a Structure

All the elements of the component tree can be read in sequence with the operations

```
start  {  nextElement  }
```

where the operations are defined as

```
void start (void);
void start (int positionNumber);
int nextElement (ComponentElement *elem, int *endOfSubcomponent);
```

The first form of the *start* operation causes the subsequent *nextElement* operations to start at the beginning of the component tree. The second form causes them to begin at the node with the given position number.

The *nextElement* operation retrieves the next element of the component tree. If the first form of the *start* operation is used, the *nextElement* operation ranges over the entire tree. If the second form of the *start* operation is used, the

nextElement operation is restricted to the subtree specified by the *start* operation's argument (if the given position number is that of a subcomponent node, it is the root node of the subtree which is searched; otherwise, the given node's parent is the root of the subtree which is searched). The type of the first argument is defined as

```
class ComponentElement {
  public:
    int id;
    ComponentElemType type;
    union {
        struct {
            int N;
            Vector *vectors;
        } vectors;                      // C_VECTORS
        QPolygon polygon;               // C_POLYGON
        Matrix4x4 *transformation;      // C_TRANSFORMATION
        char *groupName;                // C_GROUP
        char *materialName;             // C_MATERIAL
        char *label;                    // C_LABEL
        float color [RGBsize];          // C_COLOR
        float thickness;                // C_THICKNESS
        float transparency;             // C_TRANSPARENCY
        struct {
            char *name;
            int external;
        } subcomponent;                 // C_SUBCOMPONENT
        struct {
            char *name;
            int id;
        } reference;                    // C_REFERENCE
    } u;
  ComponentElement (void) { type = C_NOTYPE; }
  ~ComponentElement (void);
};
```

ComponentElemType is the enumeration

```
enum ComponentElemType {
    C_VECTORS,
    C_POLYGON,
    C_SUBCOMPONENT,
    C_MATERIAL,
    C_GROUP,
    C_COLOR,
    C_THICKNESS,
    C_TRANSPARENCY,
    C_TRANSFORMATION,
    C_REFERENCE,
    C_LABEL
};
```

QPolygon is defined as

```
typedef struct Polygon {
    int Nvertices;
    int *vertices;
    Vector *normal;
} Polygon;
```

where the *vertices* field contains indices into the current vector sequence.

The elements will be retrieved in approximately the same order in which they were specified when the component was created using the operations in the previous section, Creating a Structure. The unique position number associated with every node in the component tree is returned in the *positionNumber* field of the component

element. A component element of type C_VECTORS returns the vectors in the vector sequence. It corresponds to the calls to *startVectors* and the subsequent calls to *vector*. An element of type of C_POLYGON corresponds to a call to *polygon* and is accompanied by an element of type Polygon. An element of type C_SUBCOMPONENT corresponds to a call to *startSubcomponent* and is accompanied by the *subcomponent* element. The *subcomponent* element contains the name of the subcomponent (if it has one) and an indicator which is TRUE is it is an external subcomponent. Each call to *nextElement* returns an *endOfSubcomponent* indicator which contains the number of subcomponents (if any) that end at that point. An element of type of C_MATERIAL corresponds to a call to *material* and is accompanied by the *materialName* element. A component type of C_TRANSFORMATION corresponds to calls to *translate, rotate, scale* and *transform*. The transformations are combined into one transformation whenever possible, and the transformation is represented by a homogeneous matrix. An element of type C_REFERENCE corresponds to a call to *reference* and is accompanied by an element containing the name and position number of the referenced component and an indicator which is TRUE if it is an external component. (Note that the elements of the referenced component are *not* retrieved by calls to *nextElement*.) An element of type C_LABEL corresponds to a call to *label* and is accompanied by an element containing the label text.

The *nextElement* operation returns TRUE when a component element is returned and FALSE when the subtree specified by the *start* operation is exhausted. The various pointer fields, if not NULL, are allocated on the heap and must be freed by the caller (they are the *vectors.vectors, polygon.vertices, polygon.normal, groupName, materialName, label* and *subcomponent.name* fields).

The *start* and *nextElement* operations described above allow the component tree to be read sequentially. The following two operations allow it to be read in a random access fashion:

```
int getElement (ComponentElement *elem, int *endOfSubcomponent);
int getElement (int positionNumber, ComponentElement *elem,
                int *endOfSubcomponent);
```

The first version of the *getElement* operation returns the element at the current position of the component tree. If there is no tree or if the current position is past the end of the tree, if returns FALSE. The second version returns the element at the given position; it returns FALSE if there is no element with the given number. The *endOfSubcomponent* argument has the same meaning as for *nextElement*. After a *getElement* operation, *nextElement* may be used to read elements sequentially starting with the element after the one read by *getElement*.

An operation which is similar to *nextElement* but which returns only polygon nodes is

```
int nextElementaryComponent (ElementaryComponent *component)
```

where *ElementaryComponent* is defined as

```
class ElementaryComponent {
  public:
    int id;                       // unique identifier
    int surface;                  // TRUE if has no thickness
    int Nvertices;                // number of vertices of polygon
    Vector *vertices;             // the vertices
    Vector normal;                // normal to the polygon
    char *groupName;              // component's group
    char *materialName;           // component's material
    float color [RGBsize];        // component's color
    float thickness;              // component's thickness
    float transparency;           // component's transparency
    int tag;                      // unique QSF identifier
    ElementaryComponent (void);
   ~ElementaryComponent (void);
};
```

The *id* field is the unique identifier (position number) of the polygon node. The *surface* field is TRUE if the polygon represents a surface element which is not regarded as having thickness and FALSE if it represents an

element regarded as having thickness. The *Nvertices* field gives the number of vertices in the polygon, and the *vertices* field is a vector array containing the vertices, in order. The *normal* field contains a unit vector normal to the polygon. If the polygon represents an element with thickness, the normal vector points outward, away from the thickness of the element. The *groupName*, *materialName*, *color*, *thickness* and *transparency* fields give the attributes which are in force at the element. The *tag* gives a way of associating the elementary component (the polygon) with the QSF file (if any) in which it originated; the tag is the position number of the root node in the component tree that corresponds to the QSF file (see the *load* operation above). A sequence of elementary components may be fetched with the sequence

```
start  {  nextElementaryComponent  }
```

The *nextElementaryComponent* operation returns TRUE when a component is returned and FALSE when the subtree specified by the *start* operation is exhausted.

A similar operation that returns only the root nodes that correspond to QSF files is

```
int nextFileComponent (int *id, char **name, int *parentId);
```

The *id* argument is the position number of the root node of the subcomponent corresponding to the QSF file, the *name* argument is the subcomponent's name, and the *parentId* is the position number of the subcomponent's parent node. If there is no parent, the parent id is zero. A sequence of file components may be fetched with the sequence

```
start  {  nextFileComponent  }
```

The *nextFileComponent* operation returns TRUE when a component is returned and FALSE when the subtree specified by the *start* operation is exhausted.

Selecting Subparts of a Structure

There are two ways that subparts of a structure can be selected, by structure and by attribute. Various operations are provided to support structural and attribute selection.

Structural selection designates a particular node; the node is considered selected and is called the *selected element*. The smallest selectable element contains a single polygon and is called an *elementary component* or *eComp* for short. When a DXF file is converted and loaded (see above), an *eComp* is created for each polygon of the DXF input file.

Attribute selection designates all elementary components that possess a particular set of attributes (group, material, paintColor, paintTransparency, or thickness); those subcomponents are considered to be selected and are called the *current attribute-selected elements*. The set of attributes used for selection may contain at most one group, at most one material, at most one paintColor range, at most one paintTransparency range, and at most one thickness range. The current attribute-selected elements are those which belong to the selected group (if any), have the selected material (if any), have the selected paintColor (if any), fall in the selected paintTransparency range (if any) *and* fall in the selected thickness range (if any). SCG inheritance holds with respect to the selected group, so an element is selected by group if its group is a descendant of the selected group.

Provided the component tree is not empty, there is always a selected element (initially the root) and always a set of attribute-selected elements (initially empty).

The purpose of selecting elements is to manipulate them in some fashion, perhaps to change their attributes. It is possible to apply operations to the selected elements (see the next section), either to those selected structurally or to those selected by attribute. It is also possible to apply operations to the intersection of the current subcomponent and the current attribute-selected elements.

When operations are applied to attribute-selected elements or to the intersection of the current subcomponent and the current attributed elements, they are applied to each eComp in the set. When operations are applied to structurally-selected elements, they can be applied in two fashions, either to each eComp (as with attribute-selected elements) or at the current position in the structure. Normally, attribute changes are applied to each eComp and transform changes are applied at the current position. Attribute changes may also be applied at the current position, but doing so is inherently trickier than applying them to each eComp; the affect they have on eComps then depends on inheritance and on whether they are overridden further down in the subcomponent.

The operations that support structural selection are:

```
int select (int positionNumber);
int selectRoot (void);
int up (void);
int down (void);
int selected (void);
```

The *select* operation sets the selected element. If the argument is a valid (existing) position number, it is made the selected node; in any case, the position number of the currently selected node is returned. The *selectRoot* operation makes the root the currently selected node and returns its position number (or zero if there is no component tree). The *up* operation moves the current component to the parent of the current node and returns its position number (if the current element is the root, it remains the root). The *down* operation undoes the effect of the last previous *up* operation (provided there has been an *up* operation since the last *select* operation) and returns the position number of the new current element (if there is no *up* operation to be undone, the current element remains as it is). The *selected* operation returns the position number of the current element.

The operations that support attribute selection are given below.

```
void selectGroup (char *groupName);
void resetSelectedGroup (void);
char *selectedGroup (void);
```

The *selectGroup* operation sets the currently selected group name. The *resetSelectedGroup* operation makes the attribute selection set contain no group. The *selectedGroup* operation returns the name of the currently selected group (NULL if none); the character string (if not NULL) is allocated on the heap and must be deleted by the user.

```
void selectMaterial (char *materialName);
void resetSelectedMaterial (void);
char *selectedMaterial (void);
```

The *selectMaterial* operation sets the currently selected material. The *resetSelectedMaterial* operation makes the attribute selection set contain no material. The *selectedMaterial* operation returns the name of the currently selected material (NULL if none); the character string (if not NULL) is allocated on the heap and must be deleted by the user.

```
const int RGBsize = 3;
void selectPaintColor (float low [RGBsize], float high [RGBsize]);
void resetSelectedPaintColor (void);
int selectedPaintColor (float low [RGBsize], float high [RGBsize]);
```

The *selectPaintColor* operation sets the currently selected paint color range. The *resetSelectedPaintColor* operation makes the attribute selection set contain no paint color. The *selectedPaintColor* operation returns the currently selected paint color; it returns TRUE if there is one and FALSE otherwise.

```
void selectPaintTransparency (float lo, float hi);
void resetSelectedPaintTransparency (void);
int selectedPaintTransparency (float *lo, float *hi);
```

The *selectPaintTransparency* operation sets the currently selected paint transparency range. The *resetSelectedPaintTransparency* operation makes the attribute selection set contain no paint transparency. The *selectedPaintTransparency* operation returns the currently selected paint transparency range; it returns TRUE if there is one and FALSE otherwise.

```
void selectThickness (float lo, float hi);
void resetSelectedThickness (void);
int selectedThickness (float *lo, float *hi);
```

The *selectThickness* operation sets the currently selected thickness range. The *resetSelectedThickness* operation makes the attribute selection set contain no thickness. The *selectedPaintTransparency* operation returns the currently selected thickness range; it returns TRUE if there is one and FALSE otherwise.

The entire attribute selection set can be set, emptied or queried with the operations

```
void selectAttributes (SelectionSet selected);
void resetSelectedAttributes (void);
void selectedAttributes (SelectionSet *selected);
```

where SelectionSet is defined as

```
typedef struct SelectionSet {
    char *groupName;
    char *materialName;
    struct {
        int selected;
        float low [RGBsize];
        float high [RGBsize];
    } colorRange;
    struct {
        int selected;
        float low;
        float high;
    } thicknessRange;
    struct {
        int selected;
        float low;
        float high;
    } transparencyRange;
} SelectionSet;
```

The group name and material are considered to be in the selection set if the *groupName* and *materialName* fields, respectively, are non-NULL. The other attributes are considered to lie in the set if their *selected* booleans are TRUE.

The operations

```
int nextSelectedElement (void);
int nextSelectedElement (ComponentElement *element);
void startSelectedElements (void);
int getSelectedElements (int *N, int **positionNumbers);
```

are used to obtain the selected elementary components. The two forms of the *nextSelectedElement* operation return them one at a time. The first form returns the identifier of the next selected component or zero if the selected components are exhausted. The second form retrieves the next selected component itself and returns TRUE, or returns FALSE if the selected components are exhausted. The *startSelectedElements* operation restarts the sequence of elements returned by *nextSelectElement* to the beginning. The *getSelectedElements* operation returns all the selected elements together. The *positionNumbers* array is allocated on the heap and must be deleted by the client.

The above operations can be used to return the attribute-selected elementary components, the structurally selected elementary components, or the intersection of the two. If the attribute selection set is empty, all components satisfy the attribute selection criteria. If the current component is an elementary component, it is the only structurally selected component; if the current component is a subcomponent, all elementary components contained in it are considered to be structurally selected. If *selection intersection* is turned on, the above operations return the intersection of the structurally selected and the attributed selected elements. Selection intersection can be turned on and off by the operation

```
int selectIntersection (int on);
```

If the *on* argument is TRUE, selection intersection is turned on and it if is FALSE, it is turned off. The operation returns the previous value of the selection intersection indicator. The operation

```
int intersectionSelected (void);
```

can be used to return the current value of the selection intersection indicator. To summarize, if the attribute selection set is empty, only the structurally selected elements are returned. If the attribute selection set is not empty and selection intersection is off, only the attribute-selected elements are returned. If the attribute selection set is not empty and selection intersection is on, those attribute-selected elements that lie within the current component are returned.

Attribute Retrieval

The attributes of a component can be retrieved by means of the operation

```
char *getGroup (int positionNumber);
char *getGroup (void);
char *getMaterial (int positionNumber);
char *getMaterial (void);
void getPaintColor (int positionNumber, float color [RGBsize]);
void getPaintColor (float color [RGBsize]);
float getPaintTransparency (int positionNumber);
float getPaintTransparency (void);
float getThickness (int positionNumber);
float getThickness (void);
void getPhysicalAttributes (int positionNumber,
                            char **materialName,
                            float paintColor [RGBsize],
                            float *paintTransparency,
                            float *thickness);
void getPhysicalAttributes (char **materialName,
                            float paintColor [RGBsize],
                            float *paintTransparency,
                            float *thickness);
void getVectors (int *Nvectors, Vector *vectors);
void getVectors (int positionNumber, int *Nvectors, Vector *vectors);
```

The returned strings are allocated on the heap and must be deleted by the caller. The *getVectors* operations return the vector sequence at the current position or the position indicated by the position number argument. The vectors are allocated on the heap and must be deleted by the caller.

Related operations are

```
int centroid (Vector *v);
int centroid (int positionNumber, Vector *v);
```

The first form of the operation computes the geometric centroid of the current subcomponent, and the second form computes the centroid of a given subcomponent. The current (given) subcomponent is the smallest subcomponent containing the current (given) node.

Altering a Structure

As mentioned above, the purpose of selecting components is to be able to alter them. Several operations are provided for altering the attributes of the selected elements:

```
void changeGroupAllSelected (char *groupName);
void changeMaterialAllSelected (char *materialName);
void changePaintColorAllSelected (float color [RGBsize]);
void changePaintTransparencyAllSelected (float transparency);
void changeThicknessAllSelected (float thickness);
```

The *changeGroupAllSelected*, *changeMaterialAllSelected*, *changePaintColorAllSelected*, *changePaintTransparencyAllSelected* and *changeThicknessAllSelected* operations change, respectively, the group name attribute, the material attribute, the paint color attribute, the paint transparency attribute, and the thickness attribute of all selected elementary components. A set of attributes can be changed all at once with the operation

```
void changeAttributesAllSelected (AttributeSet attributes);
```

where AttributeSet is defined as

```
typedef struct AttributeSet {
    char *groupName;
    char *materialName;
    struct {
        int present;
        float value [RGBsize];
    } color;
    struct {
        int present;
        float value;
    } transparency;
    struct {
        int present;
        float value;
    } thickness;
} AttributeSet;
```

The group name attribute is considered to be in the set if the *groupName* field is non-NULL, and likewise with the material attribute and the *materialName* field. Note that no check is made to verify that the group name is currently the name of a structural component grouping or that the material name can currently be found in the material database (see Section 3.3). The other attributes are considered to be in the set if their *present* Boolean is TRUE.

The component tree can also be modified by inserting or altering nodes at the current position in the component tree by using the operations defined above in the section titled Creating a Structure, namely

```
void startVectors (void);
void vector (int Nvectors, Vectors *v);
void vector (Vector v);
int polygon (int Nvertices, int *vectorIndex);
int polygon (int Nvertices, int *vectorIndex, Vector normal);
void group (char *groupName);
void material (char *matName);
void paintColor (float color [RGBsize]);
void paintTransparency (float trans);
void thickness (float thick);
void label (char *text);
int startSubcomponent (void);
int startSubcomponent (char *name);
void endSubcomponent (void);
int reference (char *name);
```

```
void scale (Vector scale);
void translate (Vector translation);
void rotate (Vector axis, float rotation);
void transform (Matrix4x4 transformation);
```

These operations insert nodes only when necessary. The transformation operations (*translate*, *rotate*, *scale* and *transform*) in effect compose the appropriate transformation with the cumulative transformation at the current position. A *vector* operation adds vectors to the current vector sequence node is there is one (if there is not, a *startVectors* operation may be used to start one). The *group*, *material*, *paintColor*, *paintTransparency* and *thickness* operations merely alter the node of the appropriate type if there is one present in the current subcomponent and insert one if there is not. The *polygon*, *label*, *reference*, *startVectors* and *startSubcomponent* operations always insert a new node.

Removing Elements

Operations are provided for removing parts of the component tree:

```
int removeSubcomponent (void);
int removeLabel (void);
int removeTransformation (void);
int removeVectors (void);
int removePolygon (void);
int removeReference (void);
int removeGroupName (void);
int removeMaterialName (void);
int removePaintColor (void);
int removePaintTransparency (void);
int removeThickness (void);
```

Separate operations are provided for the different node types as a security measure, to make it more likely that the user is removing what he or she thinks she or he is. The above operations remove the node of the appropriate type at the current position in the component tree. If the current component is of the given type, it is removed. There can be at most one sibling node of type group, material, color, transparency or thickness; if the removal of a node of one of those types is requested and the current component is a subcomponent node, its child of that type (if any) is removed. If the removal of a subcomponent node is requested and the current node if not a subcomponent node, its parent is removed. When a subcomponent node is removed, its BSP tree is destroyed if it has one (see next section). Each of the operations returns TRUE if a node was removed.

There are corresponding operations for removing a node at a position in the component tree given by a position number:

```
int removeSubcomponent (int positionNumber);
int removeLabel (int positionNumber);
int removeTransformation (int positionNumber);
int removeVectors (int positionNumber);
int removePolygon (int positionNumber);
int removeReference (int positionNumber);
int removeGroupName (int positionNumber);
int removeMaterialName (int positionNumber);
int removePaintColor (int positionNumber);
int removePaintTransparency (int positionNumber);
int removeThickness (int positionNumber);
```

Computing Material Interactions

BSP trees are built by the following operations:

```
int buildBSP (void);
int buildBSP (int positionNumber);
```

The first builds one for the subcomponent at the current position. The position number of the subcomponent node for which the BSP tree was built is returned (zero if there is no component tree). The second builds a BSP tree for the subcomponent at the given position (provided the identified node is a subcomponent node) and returns the subcomponent's id (or zero if the tree is not built).

BSP trees can be removed with the operations

```
int removeBSP (void);
int removeBSP (int positionNumber);
```

The first removes the BSP tree for the current subcomponent (if it has one) and returns the id of the current subcomponent node (or zero if it had no BSP tree). The second removes the BSP tree at the given position (provided the identified node is a subcomponent node and has a BSP tree) and returns the subcomponent's id (or zero if no tree is removed).

The following operation is suitable for picking, if picking is not done by the GRE process:

```
int findNearest (Vector x0, Vector r, Vector *at);
```

The *findNearest* operation shoots a ray from point *x0* in direction *r*. It returns the position number of the first elementary component the ray encounters (zero if none). If an eComp is encountered, the location of the intersection is returned in *at*.

The *findAll* operation returns a list of all the components encountered by a ray. There are two forms of the operation:

```
void findAll (Vector x0, Vector r,
              int *Ninteractions, MaterialInteraction **interactions);
void findAll (Vector x0, Vector r,
               int *Nintersections, Intersection **intersections);
```

where MaterialInteraction is defined as

```
class MaterialInteraction {
  public:
    char *materialName;
    float thickness;
... };
```

and Intersection is defined as

```
class Intersection {
  public:
    Vector location;
    int id;
};
```

The *findAll* operation shoots a ray from point *x0* in direction *r*. The first form returns in *Ninteractions* the (nonnegative) number of material interactions and the interactions in the *interactions* array. A material interaction consists of a material name and a thickness. Both the *interactions* array and the *materialName* strings are allocated on the heap and must be deleted by the caller. The thickness value represents the thickness of the material traversed by the ray; it takes into account the angle at which the ray strikes the material, and whether the material is represented by a polygon with thickness or a polygon which is a surface element (see above). In the case of a polygon with a thickness attribute, the value also takes into account whether the ray enters and exits the material through one of the faces of the polyhedron filled with the material which is parallel to the polygon or through one of the (implied) faces which are perpendicular to the polygon. For each polygon encountered, the material is the current material at the polygon's position in the component tree, and similarly for the thickness

attribute when it is used. The second form of *findAll* returns in *Nintersections* the number of intersections that the ray makes with the polygons in the component tree and in *intersections* the position number of each polygon and the location at which the ray intersected it. The intersections are given in order, from the nearest to *x0* to the farthest.

SCG Operations

Structural Component Groupings are used in conjunction with the components in the CDB; therefore various SCG operations are provided in the CDB. There are operations for loading and storing the SCGs, for reading the SCGs sequentially, for reading a particular SCG, for defining a new SCG, for modifying an existing SCG and for removing an SCG.

Loading and Storing SCGs

The CDB initially reads the SCGs from a default file (even if the default file is not present, a group named "DEFAULT" is defined). The SCGs may be changed during the operation of the system, by the operations described below or by the addition of group names when a component file is loaded (see above). The user can write out the current SCGs with the operations

```
int storeGroups (char *filename, char *path);
int storeGroups (void);
```

The first operation writes the SCGs to the given filename in the directory given by the *path* argument. The second operation writes the SCGs to the file to which they were most recently written or from which they were most recently read. The operations return TRUE if the file was successfully stored. The user can load a new set of SCGs with the operations

```
int loadGroups (char *filename, char *path);
int loadGroups (void);
```

The first loads the SCGs from the given filename and directory. The second reads them from the file from which they were most recently read or to which they were most recently written. The operation

```
int loadDefaultGroups (void);
```

restores the SCGs from the default file. All three operations return TRUE if the SCGs were successfully read.

Reading SCGs Sequentially

The operations

```
void startGroups (void);
int nextGroup (char **parentName, char **groupName, int *level);
```

can be used to read the SCGs sequentially. The *startGroups* operation resets the current SCG position to the head of the SCG hierarchy (the default group). The *nextGroup* operation produces name of the next group in the hierarchy. Its parent's name is also returned in order to make the hierarchical structure known to the reader. The nonnegative *level* argument expresses the hierarchical relationship among the groups. There is only one group at level zero, the root group (typically named "DEFAULT"). Its children have level 1, their children level 2 and so on. The function value is TRUE if a group name is returned and FALSE is the groups are exhausted. The returned strings are allocated on the heap and must be deleted by the caller.

Reading a Particular SCG

A particular SCG can be read with the operation

```
int getGroup (char *groupName,
              char **parentName,
              char **materialName,
              float color [RGBsize],
```

```
                    float *thickness,
                    float *transparency);
```

The *groupName* argument gives the name of the group to be read. The returned arguments give the name of the group's parent, the name of the group's default material, and the group's default color, thickness and transparency. The operation returns FALSE if there is no group with the given name (in which case the returned arguments are meaningless). Because the group color alone is often desired, the operation

```
int getGroupColor (char *groupName, float color [RGBsize]);
```

is provided. The arguments have the same meaning as in the *getGroup* operation.

Creating a New SCG

A new SCG is created with the operation

```
int createGroup (char *groupName, char *parentName);
int createGroup (char *groupName, char *parentName,
                 char *materialName, float color [RGBsize],
                 float thickness, float transparency);
```

The *groupName* and *parentName* arguments give the name of the new group and the name of its parent, respectively. The other arguments, if present, give the values of the attributes for the new group. If the other arguements are not present, the new group's attributes are inhereted from its parents. The operation returns TRUE if the group was created; it fails to be created if a group of the given name already exists or if the parent does not exist.

Changing an Existing SCG

An existing SCG can be changed with the operations

```
int changeGroupName (char *oldGroupName, char *newGroupName);
int changeGroupMaterial (char *groupName, char *materialName);
int changeGroupColor (char *groupName, float color [RGBsize]);
int changeGroupThickness (char *groupName, float thickness);
int changeGroupTransparency (char *groupName, float transparency);
int changeGroupPhysicalAttributes (char *groupName, float color [RGBsize],
                                   float thickness, float transparency);
```

The *changeGroupName* changes a given group name to a new name. It returns TRUE if the name was changed and FALSE if it was not; it is not changed if the old name is not in use or the new name is already in use. When the change is made, all elements of the component tree that have the old name as an attribute are changed to use the new name. All the other operations make the indicated changes and return TRUE if the named group exists (and the change is made) and FALSE if it does not.

Removing an SCG

A group name can be removed by the operation

```
int removeGroup (char *groupName);
```

The *removeGroup* operation removes the named group if it exists; it returns TRUE if the group existed and was removed and FALSE otherwise. If the group is removed, all elements of the component tree that have the group as an attribute are changed to use the group's parent. It is impossible to remove the root group, which has the name "default".

External File Format

A subset of the Virtual Reality Modeling Language (VRML) is used as the external file format. The node types used for components and the corresponding VRML constructs are given in Table 3.4. The subcomponent name

argument corresponds to the VRML DEF keyword. The *reference* operation corresponds to the VRML USE keyword.

3.5.2 STRUCTURAL COMPONENT GROUPINGS

A Structural Component Grouping (SCG) is a hierarchy of names which can be used as attributes of the structures contained in the Component Data Base (CDB). When a component is installed in the CDB, the SCGs in use are determined by the layer names in the input DXF file and by a set of SCGs provided in an SCG file which is used as a second input file. Different SCG files can be used at different times. The SCGs are a hierarchically arranged set of group names, with physical attributes attached to the groups. The physical attributes are the material (see the Material Database section 3.3), the color, the thickness and the transparency. The entire set of group names can be thought of as being arranged in a tree with the name "Default" at the root. The Default group has, appropriately, default physical attributes to go along with it. The SCGs can be altered interactively by the QUEST user (see Section 2.1.1).

CDB Node Type	VRML Node or Field Type
subcomponent	Separator
vector	Coordinate3
polygon	IndexedFaceSet
translation	Translation
rotation	Rotation
scaling	Scale
transformation	MatrixTransformation
material, paintColor, paintTransparency	Material
thickness	A self-defining node
label	Info

Table 3.4: The subset of the Virtual Reality Modeling Language used in QUEST.

An SCG file is an ASCII file containing the group names and their associated physical attributes. Tab characters before the group name indicate the level of that group in the hierarchy. The physical attributes are given in the order *material*, *thickness*, *color*, and *transparency*. The *material* is the name of a material in the MDB. *Thickness* is the thickness in inches. The color may be one of the colors recognized by AutoCAD or an RGB triple. *Transparency* is a number in the range [0,1], where 1 means completely transparent and 0 means completely opaque. Any attribute may be given as "parent", in which case it inherits the corresponding attribute of its parent. Trailing "parent" attributes may be omitted.

Consider the following example, where Figure 3.3 expresses the simple hierarchy:

```
Door   Wood   2.0   BROWN
<TAB>   Interior Door   parent   1.5   WHITE
<TAB>   Exterior Door   Steel1   2.0    BLACK
Wall    Gypsum    4.0   (.6, .5, .4)
<TAB>  Interior Wall
```

```
<TAB>  Exterior Wall
```

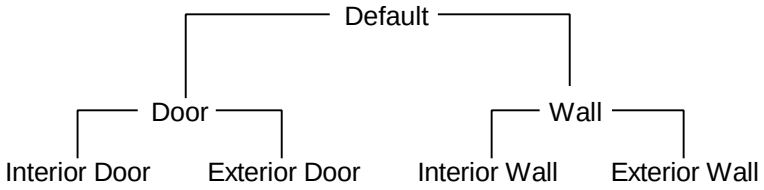


Figure 3.3: Example hierarchy of group names.

The group "Door" has material Wood, thickness 2.0, color BROWN and the transparency of its parent "Default", which is 0.0. The group "Interior Door" has attributes Wood, 1.5, WHITE and 0.0. The group "Exterior Door" has attributes Steel1, 2.0, BLACK and 0.0. The group "Wall" has attributes Gypsum, 4.0, (.6, .5, .4) and 0.0, as do groups "Interior Wall" and "Exterior Wall". The triple (.6, .5, .4) means (red = .6, green = .5, blue = .4), where the RGB values run from 0 to 1. The names are not case-sensitive.

3.6 INTERPROCESS COMMUNICATION

The Interprocess Communication (IPC) logic allows a client and a server which are parts of different heavyweight processes to communicate with each other. The client makes Remote Procedure Calls (RPC) that are received by the server. For each RPC (Birrell and Nelson 1984), the client blocks until the server informs it the call is complete. The client may pass input arguments to the server and the server may pass output arguments to the client. A server may have multiple clients.

Description

A server may be thought of as an object whose operations (member functions) can be invoked by clients. The clients can invoke the operations even if the object is remote from the clients (in a different process or on a different machine). The present implementation supports clients and servers that are in different heavyweight processes in the same machine. Future implementations will support clients and servers on different machines. However, the application-level logic will remain unchanged when this capability is added.

The operations of an object which acts as a server are defined as member functions in C++. Suppose the operations that the server will perform are defined as member functions of class ObjType. If the client and server occupied the same process, an object of class ObjType would be created to play the role of server and the client would call its member functions. The client and server might operate on different threads or on the same thread. In order to permit the client and server to occupy different processes, we will define two classes derived from ObjType, namely ObjTypeClient and ObjTypeServer. An object of class ObjTypeClient is instantiated in the client process, and an object of class ObjTypeServer is instantiated in the server process. Let the operations of ObjType be defined as virtual functions, so that they can be overridden in ObjTypeClient and ObjTypeServer. In order to invoke member function P(a,b), the client invokes it from ObjTypeClient instead of from ObjType; however, the invocation looks exactly the same.

If the client and server were in the same process, the implementation of function P(a,b) within class ObjType would merely carry out operation P, whatever it is; however the implementation of P(a,b) within class ObjTypeClient must send the request to the server for it to be carried out there. The implementation of P(a,b) within class ObjectTypeClient accordingly looks like:

- prepare input arguments *a* to be passed to server,
- send request for operation *P* to server, including input arguments,
- wait for completion of the operation,
- take output arguments passed from server,
- convert output arguments so they can be returned as *b*.

These are the same steps used in conventional remote procedure calls. In standard RPC—RPC as defined by the Internet draft standard (Internet Engineering Task Force 1994 (RPC)), which was modeled on Sun RPC—preprocessors are used which to some extent conceal these steps from the programmer. The two argument conversion steps are needed because a single untyped transmission mechanism is used for arguments of all types. Note that this replicates the semantics of the ordinary procedure call, even though the caller and the called may be on different computers. One might say that the purpose of RPC is to allow the software technology of the past (procedure calls or function calls) to live on in the non-von Neumann future. It is possible to define non-blocking calls, but there is no advantage in doing so unless client requests are queued, in which case it would be better to go to full-fledged messaging using send/receive semantics. The sends can be made to look like non-blocking procedure calls, but the underlying mechanisms are rather different.

One of the more annoying aspects of RPC programming, whether done as shown above or through standard RPC mechanisms, is the need for the programmer to define manifest constants to identify the various remotely callable procedures. In standard RPC, there is only one global "identifier space", so all identifiers must be unique (at least all identifiers in any set of programs which will be registered simultaneously as able to provide RPC services). One

advantage of the object-oriented technique described here is that the identifiers need to be unique only within the object type. A good place to define the identifiers is in the header file in which class ObjectType is declared. The outline below shows how this might be done:

```
class ObjType {
            ...
            void P0 (a0, b0);
            void P1 (a1, b1);
            ...
};
enum class ObjType_Proc {
      ObjType_P0,
      ObjType_P1,
      ...
};
```

Items of type ObjType_Proc can be used for the *procedureId* argument of the *call* function of type IpcClient (see below). Note that even though C++ distinguishes functions with the same name but different argument types (overloaded functions), it is necessary to define a separate *enum* value for each of the functions.

The server object operates on its own thread. (It would be possible to use the same thread for server and client in the same process, but this is designed to be a general mechanism that will work no matter where the client and server are.) It must await a request (such as the request for operation P above), carry it out, then wait for the next request. The server's thread is created during construction of an object of type ObjectTypeServer. The server thread performs a loop that looks like this:

```
while TRUE
      ◦ wait for a request from a client
      ◦ discover the identity of the requested operation

         switch ( operation )
            .....
            case P:
            ◦ convert transmitted input arguments to typed arguments a
            ◦ invoke function P(a,b) of class ObjectTypeServer
            ◦ prepare output arguments b for transmission to client
            ◦ inform client the operation is done, passing it the
                  output arguments
            .....
      end switch
end while
```

Class ObjTypeServer contains two additional member functions *accept* and *reply*, which enable the server to wait for a request from a client and inform the client the operation is done, respectively. From this, we can see that the implementation of operation P(a,b) in ObjectTypeServer is exactly the same as it would have been in ObjectType in a single process system. Therefore, in an object-oriented scheme, the server can inherit the operations (such as *P(a, b)*) from type ObjType. The steps performed under "case P" are the same steps used in conventional remote procedure calls. Again, in standard RPC, preprocessors are used which mostly conceal these steps from the server programmer. Also, in standard RPC, logic is provided by the underlying system which takes care of the function performed by the part of the *while* loop outside "case P".

It should be apparent from the above that RPCs cry out for automation. The steps outlined above for client and server could be carried out by a simple compiler or preprocessor, something like the *rpcgen* processor associated with standard RPC (*rpcgen* Manual Page). The uncertain state of standard RPC on Windows NT rules out its use at present. It is not clear that the performance of standard RPC would be adequate on a single-processor system.

Intercommunication Mechanism

A particular server and its client communicate through a data structure kept in shared memory. There is just one instance of the structure, used by the server and all the clients. In addition to a memory area used to pass the input arguments from the client to the server and the output arguments from the server to the client, the structure contains data items (system dependent) for achieving exclusive control of the data structure, for allowing the server to wait for a request, and for allowing the client to wait for the completion of a request.

The interfaces to the facilities for interprocess communication are defined in the header file *ipc.h*. In particular, it defines structure *IpcBuffer* (the shared data structure), base class *Ipc* and derived classes *IpcClient* and *IpcServer*. Mutual exclusion and synchronization are achieved by means of semaphores (Dijkstra 1968). There is a largish untyped shared storage area for input and output arguments, which are passed back and forth as sequences of bytes. The derived type *IpcClient* contains member function *call*, through which a client issues a remote procedure call and the data conversion functions *serialize* and *deserialize*. The calling sequence of *call* is

```
void call (int programId,
           int versionNumber,
           int procedureId,
           const unsigned char *inputArguments,
           unsigned int inputLength,
           unsigned char *outputArguments,
           unsigned int *outputLength);
```

programId and *versionNumber* are unused but are present for compatibility with the Internet RPC standard. *programId* identifies which of the member functions of ObjectType is being called. *inputArguments* is the input arguments, serialized into a stream of bytes. *inputLength* is the length in bytes of the input argument stream. *outputArguments* is the output arguments, serialized into a stream of bytes. *outputLength* is the length in bytes of the output argument stream.

The *serialize/deserialize* functions convert data values to/from internal machine-dependent format to External Data Representation (XDR) (Internet Engineering Task Force, 1994 (XDR)). The calling sequences of the *serialize* routines are

```
unsigned char *serialize (int value, unsigned char *xdrValue);
unsigned char *serialize (float value, unsigned char *xdrValue);
unsigned char *serialize (double value, unsigned char *xdrValue);
unsigned char *serialize (char *value, unsigned char *xdrValue);
```

value is the value to be serialized, *xdrValue* is the serialized XDR, The returned value points to the next byte after the serialized value in *xdrValue*.

The calling sequences of the *deserialize* routines are

```
unsigned char *deserialize (const unsigned char *xdrValue, int *value);
unsigned char *deserialize (const unsigned char *xdrValue, float *value);
unsigned char *deserialize (const unsigned char *xdrValue, double *value);
unsigned char *deserialize (const unsigned char *xdrValue, char *value,
                            const int maxSize);
```

xdrValue is the XDR value, *value* is corresponding deserialized value, *maxSize* is the maximum size of the string value, including the ASCII NUL terminator.

It is assumed that all compound data types used in this application are ultimately made up of *int*, *float*, *double* and *string* values, so *serialize/deserialize* routines for compound types can be constructed from the above four routines. For example, consider

```
// DEFINITION OF TYPE T
const int Xsize = 4;
```

```
typedef struct T {
      int a;
      float x[Xsize];
} T;

// SERIALIZE VARIABLE OF TYPE T
unsigned char *serialize (T value, unsigned char *xdrValue) {

      unsigned char *p;   // BUFFER POINTER
      int i;              // LOOP INDEX

      // INITIALIZE OUTPUT POINTER
      p = xdrValue;

      // SERIALIZE COMPONENT a OF T
      p = serialize (value.a, p);

      // SERIALIZE EACH ELEMENT OF COMPONENT x OF T
      for (i = 0; i < Xsize; i++)
            p = serialize (value.x[i], p);
}
```

The derived type *IpcServer* contains the member functions *accept* and *reply*, through which the server waits for a request and informs a client the request is done, respectively; and the data conversion functions *serialize_int*, *serialize_float*, *deserialize_int* and *deserialize_float*. The calling sequences of *accept* and *reply* are

```
int accept (void);
void reply (void);
```

The *accept* function returns the operation code passed by the client (argument *procedureId* of *call*).

The calling sequences of the *serialize* functions are

```
void serialize (int value);
void serialize (float value);
void serialize (double value);
void serialize (const char *);
```

value is a value to be serialized into XDR format. The XDR is placed at the current position in the shared *IpcBuffer* and the current position is incremented accordingly.

The calling sequences of the *deserialize* functions are

```
void deserialize (int *value);
void deserialize (float *value);
void deserialize (double *value);
void deserialize (char *value, int maxSize);
```

value is the value that results from the deserialization of the XDR value at the current position in the shared *IpcBuffer*; the current position is incremented accordingly. *maxSize* is the maximum size of the output string value, including the ASCII NUL terminator.

The discussion so far applies to a single-threaded server. A single *IpcBuffer* is sufficient for a single-threaded server; a server with *N* potential threads could be implemented using an array of *N IpcBuffer* structures. Note that a single-threaded server can accept any number of clients but cannot overlap its execution of the operations they request.

3.7 GRAPHICS RENDERING ENGINE

The Graphics Rendering Engine (GRE) consists of two three-dimensional (3D) windows: the top-down view and the first person point of view. The top-down view allows the user to see his/her current view position within the graphics "world" from an overhead view position. The first person point of view allows the user to see the graphics "world" from an immersed point-of-view. An arrow marks the current position within the overhead view. The center of the arrow is the current view position within the graphics world. The direction of the arrow is the direction the user is facing within the first person view window.

Each graphics window has means to move the view position. The top-down view position can be raised or lowered only. It always "looks down" on the first person view position. The user can zoom in and zoom out on the current position. The user is allowed to zoom in to what is the equivalent of approximately two feet above the current view position. The zoom out distance is unlimited. While the user holds down the keyboard letter 'd', the view position will zoom down. While the user holds down the keyboard letter 'u', the view position will zoom up.

The first person point of view has two modes of movement defined by whether the left or right mouse button is depressed. With the left mouse button depressed the view position can be rotated left, rotated right, moved forward and moved backward. With the right mouse button depressed the view position can be panned up, panned down, panned left and panned right.

The user should think of the first person view window as being divided in to sections with a line that divides the upper half from the lower half, and a line that divides the left half from the right half for purposes of first person view position movement. With the left mouse button depressed and the cursor is in the upper half of the first person view window, the first person view position will move forward in the graphics scene. With the left mouse button depressed and the cursor is in the lower half of the first person view window, the first person view position will move backward. With the left mouse button depressed and the cursor is the left half of the first person view window, the first person view position will rotate to the left. With the left mouse button depressed and the cursor is the right half of the first person view window, the first person view position will rotate to the right. The closer the cursor is to the outside edge of the first person view window, the greater the rate of movement or rotation in that direction will be. With the cursor at the center of the first person view window, there will be no movement or rotation.

To move through a building, start with the cursor near the centerline of the first person view window. Depress and hold down the left mouse button and move the cursor toward the top edge of the first person view window. If the cursor is moved off the vertical center line of the first person view window while moving forward, the view position will continue to move forward and move to the left if the cursor is in the left half of first person view window or move to the right if the cursor is in the right half of first person view window. To "straighten out" the movement path, move the cursor back to the vertical center line of the first person view window. Moving in the reverse direction, backwards, is similar to moving forward except the cursor must be in the lower half of the first person view window.

If the view position is not inside a building, the view position elevation can be changed. By depressing the right mouse button and moving the cursor to the upper half of the first person view window the view position will pan up (move up). By depressing the right mouse button and moving the cursor to the lower half of the first person view window the view position will pan down (move down). If the view position is inside a building, the user can use the up or down arrow keys on the keyboard to change his/her view position to the next upper or next lower level in the building.

The user can pan left (move laterally to the left) or pan right (move laterally to the right) by depressing and holding down the right mouse button and moving the cursor either to the left half or the right half of the first person view window.

3.7.1 STRUCTURES

The creation of a structure for use in the QUEST project should be made by using the computer-aided design (CAD) software program AutoCAD or AutoCADLt by AutoDesk, Inc. The completed drawing must be saved as a Drawing Interchange File format (DXF). While other CAD packages exist that can save drawings in DXF format, they do not create the type of elements needed by QUEST. QUEST will only accept drawings made up of 3DPOLYs and 3D faces.

Level Isolation

For the QUEST Project, it has been decided that for each multi-storied building, each level of the building will be drawn and saved to separate DXF files. This not only makes it easier to create, copy, and edit building drawings but also allows for easy level isolation within the 3D graphics rendering. The main purpose of level isolation within the first person 3D graphics rendering is to minimize the number of polygons that need to be rendered on each 3D graphics refresh cycle. By minimizing the number of polygons that need to be rendered, a faster graphic refresh cycle can be accomplish, thus giving a smoother appearing first person 3D graphics display. Another purpose of a level isolation is to allow an overhead/topdown view of the current level in a separate graphics window. This allows the user to see where in the current level she/he is. Unlike other graphics rendering languages such as GL or OpenGL, World ToolKit (WTK) does not provide a simple method of rendering the same 3D graphical object multiple times in different graphics windows (see One Universe Graphics vs. Individual Window Parameters). (Please note, unless otherwise stated, identical WTK methods of operation will be used in both the Silicon Graphics and Windows NT operating systems.)

Method of Grouping Levels

The data that make up the 3D graphical objects is received from the CDB portion of the simulation program. In the 3D graphics portion of the simulation, the smallest entity will be a WTK object which consist of one 3D polygon. Each 3D graphical object contains a user-defined data structure. Included in the data structure is a data field that contains the graphical object's unique identification number. The unique identification number is provided by CDB. Each level of a building will consist of a group of objects. Each building will consist of a linked list of pointers to groups of levels. The collection of buildings will consist of a linked list of buildings. Dynamically building polygons and WTK graphical object is not easy and not intuitive. Below is a listing taken from WTK v2.0 Reference Manual of the necessary steps to take in creating a single WTK graphical object:

1. Initialize the object by calling WTobject_begin.
2. Add vertices to the object using WTobject_addvertex.
3. Add polygons to the object. For each polygon:
   a. Call WTpoly_begin.
   b. Add vertices (that have already been added to the object with WTobject_addvertex) to the polygon using WTpoly_addvertex.
   c. Call WTpoly_close
4. Call WTobject_close.

Note that only one object can be constructed at a time. You must complete the definition of one object before beginning the definition of a new object (no objects embedded in objects). The following example taken from QUEST's GRE illustrates the use of the object constructor functions:

```
void  LoadBuilding (char *building) {

    int                         i, curtag, receiveqsftag;
    int                         count = 0;
    int                         ubcount = 0;
    float                       xzy[3];
    BOOLEAN                     ok = TRUE;
    BOOLEAN                     okconstructobj = TRUE;
    WTobject                    *constructobj;
    WTpoly                      *constructpoly;
    WBObjectData                *newObjData;
```

```
WTgroup                      *grp = NULL;
ElementaryComponent          *eComp;

curtag = -999;

if (strlen(building) > 0)  {
   sscanf(building,"%d",&receiveqsftag);
   component->start(receiveqsftag);
}  else {
   receiveqsftag = component->makeRootCurrent();
   component->start(receiveqsftag);
}

// GET FIRST eComp
eComp = new(ElementaryComponent);

// WHILE THE CDB HAS eComps RECEIVE THEM.
while (component->nextElementaryComponent(eComp)) {

   okconstructobj = FALSE;
   ok = TRUE;

   // POLYGON MUST HAVE AT LEAST THREE POINTS
   if(eComp->Nvertices  < 3)
        ok = FALSE;

    else {

        // INITIALIZE A NEW, EMPTY OBJECT .
        constructobj = WTobject_begin();
        okconstructobj = TRUE;

        // GET VERTEX POINTS FROM THE eComp.
        for (i = 0;(i < eComp->Nvertices) && (ok == TRUE);i++) {
            xzy[X] =  eComp->vertices[i].x[X];
            xzy[Y] =  eComp->vertices[i].x[Y];
            xzy[Z] =  -eComp->vertices[i].x[Z];

            // ADD VERTICES TO THE OBJECT
            if((ok == TRUE) && ((WTobject_addvertex(constructobj,
                                 WTvertex_new(xzy))) == FALSE) ) {
                printf("addvertex error\n",eComp->id,count);
                ok = FALSE;
            }
        }

        if (ok == TRUE) {

            // BEGIN CONSTRUCTION OF A POLYGON
            constructpoly = WTpoly_begin(constructobj);

            //  ADD VERTICES PROVIDED BY eComp TO A POLYGON
            for (i = 0; (ok == TRUE) && (i < eComp->Nvertices); ++i)
                WTpoly_addvertex(constructpoly,i);

            if(ok == TRUE) {

                // “WRAP UP” THE OBJECT BY CLOSING, MAKING BOTH SIDES
                //   OF THE OBJECT VISIBLE, SETTING COLOR, AND SETTING
                //   THE OVERALL VISIBILITY OF THE OBJECT.
                WTpoly_close(constructpoly);
                WTpoly_setbothsides(constructpoly,TRUE);
                WTobject_setrgb(constructobj,(char)(255*eComp->color[0]),
                                (char)(255*eComp->color[1]),
                                (char)(255*eComp->color[2]));
                WTobject_close(constructobj,FALSE,TRUE);
                WTobject_add(constructobj);
```

```
                    WTobject_setvisibility(constructobj,TRUE);

                    // GET A NEW USER-DEFINED OBJECT DATA STRUCTURE
                    newObjData = (WBObjectData*)malloc(sizeof(WBObjectData));

                    // ASSIGN UNIQUE OBJECT IDENTIFICATION NUMBERS
                    //   PROVIDED BY THE CDB
                    (*(newObjData)).id = eComp->id;
                    (*(newObjData)).tag = eComp->tag;

                    // INITIALIZE SELECTED FIELD TO FALSE
                    ((newObjData)).selected = FALSE;

                    // ASSIGN/ATTACH THE USER-DEFINED DATA STRUCTURE
                    //   TO THE OBJECT
                    WTobject_setdata(constructobj,(void*)newObjData);

                    // FIND THE GROUP THAT THE CURRENT eComp SHOULD
                    //   BE ADDED TO OR A NEW GROUP WILL BE CREATED
                    //   FOR THE SPECIFIC eComp IDENTIFICATION NUMBER
                    if ((curtag != eComp->tag) || (grp ==NULL)){
                        grp = WBgroupGetToLoad(eComp->tag,receiveqsftag);
                        curtag = eComp->tag;
                    }

                    // IF GROUP, grp, IS NOT NULL, ADD IT TO THE GROUP
                    //   ELSE DELETE THE OBJECT--
                    if (grp != NULL)
                        WTgroup_addobject(grp,constructobj);
                    else
                        WTobject_delete(constructobj);
                }
            }
        }

        // IF UNSUCCESSFUL ON LAST eComp, DELETE THE PREVIOUS
        //   ATTEMPTED CONSTRUCTED OBJECT AND eComp AND
        //   AND TRY AGAIN ON THE NEXT eComp
        if((ok == FALSE) && (okconstructobj == TRUE))
            WTobject_delete(constructobj);
            delete(eComp);
            eComp = new(ElementaryComponent);
        }
    }

    // CLEAN-UP: DELETE LAST UNUSED eComp
    delete(eComp);
}
```

As one can see from the above example, the data coming from the CDB needs to be received in a specific order or stored so it can be used in a specific order to create WTK 3D graphical objects. The GRE needs to know how many vertices are in each eComp before it can construct a polygon and how many polygons there will be in an object. Other data needed are the color of each polygon and the polygon/graphical object unique identification number.

As each graphical object is completed, it will be added to the group of 3D graphical objects that make up a level of a building. The eComp tag, eComp→tag, has the same value for all components originating in the same DXF file, therefore a change in its value signals the change of level of a building or a different building is being loaded.

Mechanics of Isolating Current Level

As the user navigates through the first person view, the current view position is checked with the following WTK function call:

```
WTviewpoint_getposition(CurView,CurPos);
```

This returns the current X, Y and Z position within the current view (CurView) to the WTK defined variable, WT_p3, which is an array of size three of type float. The following function call is made with the current position:

```
CurF = InsideBuilding(CurPos)
```

If the current position is outside the defined boundariesof all the buildings a NULL pointer is returned and all the levels of all the buildings are graphically rendered. The boundaries of the building are kept in a data structure associated with each building. The boundaries are defined by the region within the minimum and maximum X, Y, and Z verticies. If the current position is within the defined boundaries of any building, a pointer is returned to the group of 3D graphical objects that make up the level of the building. Because the ceiling of the current level is made up of the floor of the level above, the 3D graphical objects that make up the level above are also rendered in the first person view. To find the level above the current level, the following function call is made:

```
AutoChangeFloor(CurF, CurPos)
```

If there is a level above the current level (if the user is on the roof, there won't be a level above), the group of 3D graphical objects that make up the current level and the level above are set so that they are made "visible" (to be graphically rendered) and all others are made "invisible" (not to be graphically rendered) within this function. Once it has been determined that the current position is inside the boundaries of a building and the current level has been determined, a copy of the current level is made containing all its 3D graphical objects. The copy is translated to a very distant region of the WTK universe (see One Universe Graphics vs. Individual Window Parameter Graphics below) and a second view position and view direction is created to look down at the current view position. The copy is then displayed in the overhead view window. If the user changes levels or exits the boundaries of the current building, the copy of the building is deleted and the overhead view position is set to look down at the current position and all of the buildings in the WTK universe are made visible (graphically rendered).

One Universe Graphics vs. Individual Window Parameter Graphics

WTK suffers from having only one "universe" for all of the 3D graphical windows that are rendered by a single WTK executable program. In contrast, in GL and OpenGL the user can have two or more 3D graphical windows open, each of which may have the same world coordinate boundaries. A 3D graphical object may be rendered in any one and not rendered in the other(s). All the 3D graphical objects that make up an entire building could be displayed in one window and only the 3D graphical objects that make up the current level could be displayed in the second window. No copy of the 3D graphical objects that make up the current level would need to be made nor translated to some distant part of the graphical universe to "hide" it from the main part of the graphical universe. This copying and translating to give different views of the same object not only takes care in implementing and time to accomplish but also takes up extra memory allocated for the copied 3D graphical objects.

3.7.2 PATH SPECIFICATION

Within the QUEST simulation, every active object, of which there are two types (detector and source), has an associated path. This path defines the time based movement of the active object through the synthetic environment. Issues of path definition, static representation, and interpretation are crucial to both the GRE and the TP. The path definition mechanism provides a standard set of manipulation routines, including create, delete, copy, and modify.

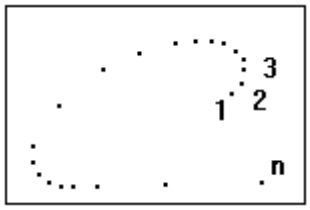
Path Creation

There are three conceivable means by which a user could initially create a path in the QUEST application: (1) first-person movement through the synthetic environment using the first-person-point-of-view (FPOV) graphics window, (2) picking an ordered series of points in the overhead graphics window which are connected together with straight line segments, and (3) formalized point specification within an input DXF structure file. Phase I of the QUEST implementation supports methods 1 and 2. Method 3 will be left to follow-on implementations, where

support for path interpretation from an input DXF file may be used to eliminate the need for a GRE during path creation. To use the two path creation mechanisms, the user must preselect a single default height above ground (above the default Z for any given location), which is applied uniformly throughout the path's definition.

Immersive Path Creation

Prior to initiating the creation of a path, the user must select the starting node for the path. The user would then commence path definition by moving through the envionment. Since the path is defined by the interactive movement of the user through the synthetic environment, path speed and movement is governed by the direct actions of the user.

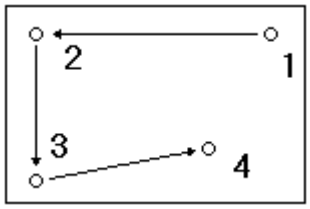


The path is defined by a WTK data structure (referred to as $path_{wtk}$), which is an ordered set of nodes, each containing two points (x,y,z), one for position and one for orientation. The density of these position nodes reflects the relative speed of the active object at that point on the path; the closer two nodes are together, the slower (in relative terms) the object was traveling at that point.

On vertical transition from one floor to another, a straight line of movement is made from the source floor to the destination floor. This is enforced by the path creation mechanism by placing a node at the point of vertical transition on the source floor, and another node directly above or below that point on the destination floor. Within the GRE, the transition from one floor to another is represented by a controlled, fluid motion. Once the interface is notified that the user would like to move up or down one floor, the GRE slowly transitions the FPOV from the source floor, through all intervening materials, to the destination floor. During this transition, the GRE display of the FPOV starts out slowly, accelerating through the intervening floor and ceiling, and then slows to a stable position directly above or below the original source point.

Point Picking Path Creation

In this mode the user selects points from a bird's eye view of the environment. Furthermore, the system assumes a default path speed. Using the 3D graphics window, the user selects a starting node for the path, enters path definition mode, and selects successive path points by orienting the cursor and pressing the keyboard space bar. Once complete, this ordered set of path points are interpreted as connected by straight line segments. The path points, together with connecting line segments, are interpreted at the default path record speed, and replaced by an ordered series of path nodes generating the $path_{wtk}$ data structure. In this path creation mode, orientation is calculated as the directed vector connecting two successive position nodes.



As a subcase of the point picking path definition method, a user can also define a specialized path consisting of a single point. Known as a *single point path creation*, such a path represents a single stationary location with a default, established orientation. An object with this path assignment will remain in the one location throughout its active assignment in the simulation system. A single point path is defined through a successive selection of the record and stop buttons during the normal immersive path creation mechanism.

Vertical transition from one floor to another is handled in the same manner as presented in the section above—straight line movement is made from the source to the destination floor.

Path Copy

The user can select a path from a list of available path names through the GUI interface, to be duplicated using the "Save As ..." button. Once duplicated, the path copy becomes its own unique path entity, and must be assigned a unique name. This facility provides the user with a means of attempting path modifications without destroying the original path.

3.7.3 STRUCTURAL COMPONENT PICKING

Structure component picking (picking) is a mode set in GRE by the user through the GUI to allow the user to pick a list of 3D graphical objects. The user may initiate picking for structural component groupings. The GRE returns a list of unique graphical object identification numbers to the TP.

Method of Picking

Once the picking mode is set from the GUI, the user may move the cursor with the mouse so that the "hot spot" of the cursor is located over/on the graphical object. The user may then press the space key on the keyboard to select the graphical object. If a previously selected object is selected again, it will be "deselected" and its "selected" flag will be set to FALSE. A method of "graphically marking", the selected object is used so the user may see the graphical objects selected (see Considerations for Marking).

Mechanics of Picking

All 3D graphical objects loaded through the CDB have a user-defined data structure attached to them. Among other fields in the data structure is a boolean field called "selected". This field is used to determine if the graphical object is "selected." When initialized this field is set to false (not selected). When the GUI initiates picking, a "picking mode" flag in the GRE will be set to true (initial default value is false). The mode flag will remain true until the GUI signals the GRE that the user has either canceled picking or has accepted the current list of selected 3D graphical objects. When either is signaled, the flag will be set to false. If the selection is accepted the GRE sends the CDB the unique identification numbers of all the selected objects.

Within the 3D graphics the following code is used to pick a graphical object:

```
rawmousedat = (WTmouse_rawdata*)WTsensor_getrawdata(Mouse->sensor);
obj = WTuniverse_pickobject(rawmousedat->pos);
```

The function call, WTsensor_getrawdata, returns the sensor-specific (mouse) raw data structure. The raw data structure for the mouse contains the current 3D location of the cursor.

The function call, WTuniverse_pickobject, returns a pointer to the 3D graphical object located at the location of the cursor or returns a NULL pointer if there is no 3D graphical object at that location. If the 3D graphical object that is pointed to by the return of WTuniverse_pickobject does not have its "selected" flag set to true (selected), it will be set to true otherwise it is set to false (not selected).

Considerations for Marking

Marking a selected graphical object allows the user to easily identify objects that have been selected. The color of marked objects is changed to a user-defined color representing a structural group. In addition, a selected object is changed a solid-filled color to a wire-framed object.

3.7.4 LIGHTING

The GRE display ambiance is a set of World ToolKit functionalities used in QUEST's to change the visual appearance of the background and structural components in the first-person view and the overhead view windows. These functionalities allow the user to change the background color and adjust ambient lighting. Settings for the display ambiance are changed through the GUI. To adjust the background color, the user first pulls down the GUI window "Image" and then selects "Background Color". A pop-up window appears with a slider bar and an input window with the current value for each of the three RGB (RED, GREEN, BLUE) values that make up the background color. There is a small, RGB display window in the GUI displaying the current background color. As the values of RGB colors are changed, the representative color will be displayed in this window. The user may either move a slider or type in a value in the input window for the appropriate RGB value. Legal value ranges for each color are 0 to 15, with $16^3 = 4096$ colors available. When the user is satisfied with the new RGB color mix, selection of "accept" applies the RGB mixture to the GRE display background.

To adjust the ambient light, the user must first pull-down the GUI window “Image” and then select “Lights”. A pop-up window will appear with a slider bar. The value range for the slider will be from 0% to 100% (no light to 100% light). The user may slide the slider bar to adjust the corresponding light value. The effects of the new values will be displayed in the GRE displays immediately.

Display Ambiance Interface and the GUI

The GRE is a server to the GUI. A lightweight process, or thread, in the GRE awaits input from the GUI. The GUI sends character strings that contain commands for the GRE to perform. For example, if the GUI signals to change the background color the following command string is sent:

```
“BACKGROUNDCOLOR  r g b”
```

The above string is first parsed in the GRE for the command “backgroundcolor”, then integer values for “r g b” are parsed from the string. In a similar fashion, the command to change the ambient lighting is the command string:

```
“AMBIENTLIGHT al”
```

The above string is first parsed in the GRE for the command “ambientlight”, then the integer value for “al” is parsed from the string. The value range for “al” is from 0 (zero) to 100. Whole number values are sufficient because incremental changes of light less than 1% are not noticeable. The following code is used to parse the above commands and perform their operations in the GRE:

```
void DoGuiCommands(char *commnd,int *threecolr, WTlight *curlight,
                   float *dirlight, float *amblight) {

      short  bgcolor;
      int    r, g, b;
      int    lightval;
      float  checklight;

      if (strstr(commnd,”BACKGROUNDCOLOR”)) {

            // PARSE commnd FOR INTEGERS r,g, and b
            ParseForRGB(command,&r,&g,&b);

            // threecolr IS A “STATE VARIABLE”
            *(threecolr+0) = r;
            *(threecolr+1) = g;
            *(threecolr+2) = b;

            // CONVERT r, g, b TO HEXIDECIMALS
            bgcolor = GetColorAdjust(threecolr);

            // WTK LIBRARY FUNCTION TO SET BACKGROUND COLOR
            WTuniverse_setbgcolor(bgcolor);

      } else if(strstr(commnd,”AMBIENTLIGHT”)) {

            // PARSE commnd FOR INTEGER light
            ParseForLight(command,&light);

            // amblight IS A “STATE VARIABLE”
            *amblight = (float)(light/100.0);

            // WTK LIBRARY FUNCTION TO SET AMBIENT LIGHT
            WTlight_setambient(ambient);
      }
}
```

3.8 GRAPHICAL USER INTERFACE

The Graphical User Interface (GUI) was developed using the portable windowing library from XVT Software (1994). XVT was chosen to satisfy several QUEST GUI development requirements, including portability across UNIX and personal computer platforms, clean C++ object organization, availability of a rich set of visual components, an easy to use graphical layout tool, and no run-time licensing costs. The XVT application framework consists of three different levels:

- An application level that controls the program,
- A document level that gets access to data and stores and manages data,
- A view level that provides windows and other specialized structures in which to gather and display data and to manipulate graphical objects.

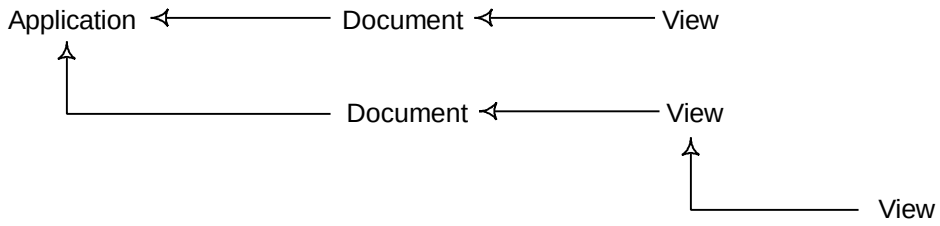


Figure 3.4: The XVT application framework.

Figure 3.4 illustrates the relationship between these three levels. The application object manages the flow of the entire application, initializes the startup environment, and responds to application events. This object creates the document levels. Access to global objects, including the desktop global that manages screen window layouts, is handled at this level. Before terminating the application any cleanup that is required is handled by the application. The document object is responsible for accessing and managing data. This object manipulates files and internal pieces of data. It acts as the link between the application and the views of the data. A document cannot display data itself, so it instantiates windows in which the data may be viewed.

The view hierarchy is comprised of all of the classes that display some form of object on the screen when they are instantiated. Views can supply native controls like buttons, check boxes, scrollbars, list boxes, etc. These controls take on the look and feel of the native window system in which the application is running. Views can be nested within another view. Views can display icons, drawing shapes, spreadsheet type grids, scrollable lists, text, sketching areas, and two and three dimensional graphics display areas.

QUEST Design Using XVT

The XVT QUEST application object controls the flow of the entire QUEST application. QUEST has been organized into a number of XVT document objects. Each document object represents a separate type of QUEST file. There are eight different types of files that can be created or manipulated from the QUEST GUI documents. The overall structure is indicated in Figure 3.5. Here DocQStr is used to manipulate structure files, DocQPath is where paths through the structure(s) are stored, DocQSrc is used to store source information, DocQDet is used to store detector information, DocQMat is where the material database information can be manipulated, DocQBack is where the background radiation information is stored, and DocQOut is where output generated from a QUEST simulation run is stored. In addition, DocQConf holds the configuration information for a given scenario—it keeps track of all the files used in a given scenario.

Under each document object in the XVT hierarchy are the views or windows that are used in QUEST to obtain information from, or to display information to, the QUEST user. The views or windows under a particular document object are specifically related to that document. Each view icon represents a separate window except for

under the document object DocQPath, where views QPathPnt, QPathImm, QPathPre, and QPathPlay represent different scenes that are displayed based on a radio button selection. For example if the Immersive radio button is chosen, QPathImm's view is displayed in a portion of the QPath window.

The top level window is QConfig. QConfig, QConStr, QConSrc, and QConDet all have large icon buttons that bring up other windows in a hierarchical fashion. The icons have a status field above them that indicates the current status of that icon: Complete/Incomplete or Ready/Not Ready. From the top-level window only one branch of logic is visible at a time. These include Structure, Source, Detector, Background, Simulation, or Analysis. If a number of windows have been opened through the use of the Structure button and its corresponding windows, and the Detector button is clicked, all the Structure windows are closed so Detector definition can be completed. This helps to minimize confusion by isolating separate sections of logic and the manipulation of their corresponding files.

Each document object has only one window associated with it that allows file manipulation commands such as New, Load, Save, and Save As. All windows that end in File (e.g. QStrFile) are pop-up dialog windows that display the file path and name, a QUEST unique name, and a description. These windows are brought up whenever a file Load, Save, or Save As command is invoked. In the case of Save or Save As the user can input a new or change an old file path and name, QUEST unique name, and description.

Separation Between the GUI and Underlying Objects

The QUEST GUI code is logically separated from all other underlying objects and their associated code. The GUI code is strictly used to gather data from the user and display data provided by the underlying objects. It is not a repository for the information and does not perform calculations. All other actions beyond getting information from the user and displaying information for the user is handled by distinctly independent objects.

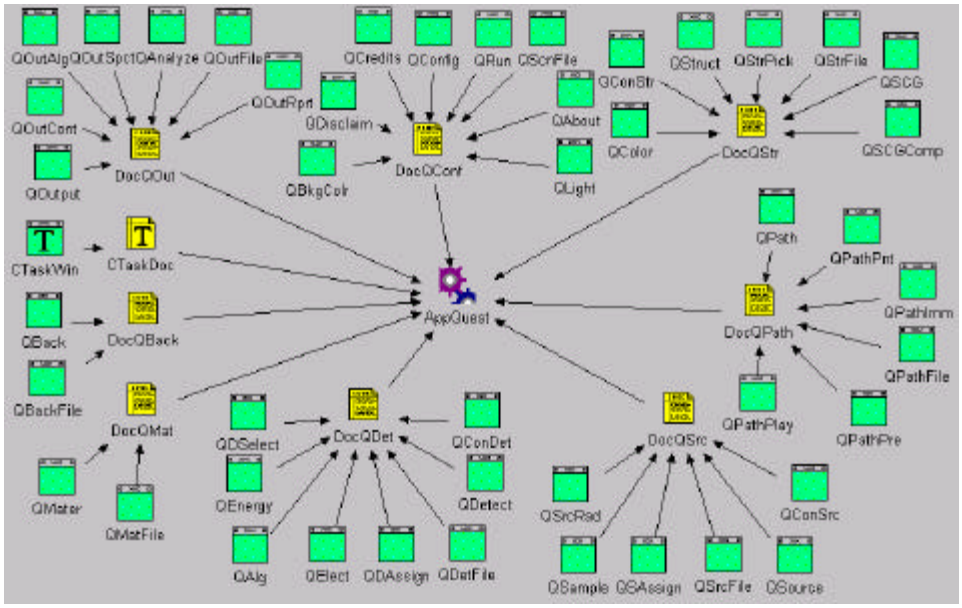


Figure 3.5: The QUEST XVT user interface blueprint.

The Source Definition window is a good example of this separation. When the user wants to create a new Source this window is brought up. The GUI fills the Element pull-down menu by querying the Periodic Table object to obtain a list of Element object names. The GUI obtains the Isotope, Quantity, and Gamma Ray table information for the first element in this list by querying the Element object. The GUI inserts Element information into the Radionuclides table when the user clicks on the Insert Element button. The Atomic number is acquired by querying the Element Radionuclide object. When the user completes the definition of a source they exit this

window and Save the source. The Source object handles its own file output, using information the GUI passes it about path, filename, QUEST name, and description.

As shown in this simple example, the GUI handles the display of list and table information, responding to user button clicks, user list selections, and user menu selections. The information is stored and retrieved from the Periodic Table, Element, Radionuclide, and Source objects. These objects are separate entities that exist independent of the GUI, maintaining strict separation of program logic and the user interface.

4 A COMPARISON OF MEASURED AND SYNTHETIC RESPONSE FUNCTIONS

In this section we present a comparison of measured and synthetic gamma-ray detector response functions. The results of the validations study (first presented in Johnson et al. 1997) compare synthetic Sodium Iodide (NaI) and Germanium (Ge) detector responses generated by QUEST, those generated by another gamma-ray detector computational model, SYNTH, and those generated by real detectors deployed in the field. Quantitative models like QUEST are important since they, (1) allow inspection teams to maximize the probability of finding materials of interest, (2) aid in the development of new instruments and detection techniques, and (3) support other diverse applications including environmental monitoring, nuclear facilities inspections, and radiation safety responder training.

It is necessary to validate QUEST against laboratory and field tests to allow users to have confidence in its results. Validation can also be attained by comparing QUEST to other codes, such as SYNTH, that are regarded as standards. We present the results of two series of validation tests. In the first, spectra generated by QUEST were compared to spectra collected in the laboratory at PNNL and to spectra generated by the SYNTH program. In the second, detector algorithm output generated by QUEST was compared to output of the same algorithms run against data collected under field conditions at the Remote Sensing Laboratory in Las Vegas. Comparison of the results is favorable in both cases.

4.1 COMPARISON OF SPECTRA

Spectral data were collected at PNNL and compared to spectra generated by QUEST runs that simulated the laboratory conditions. The QUEST spectra were also compared to spectra generated by the SYNTH software.

The data acquisition system used to collect the laboratory spectra was very simple, and used off-the-shelf Nuclear Instrument Modules (NIM), and commercially available software. A standard NIM bin was used to house a HV-bias supply [for the PMT], a spectroscopy grade linear amplifier, and an ORTEC ADCAM Multi-Channel Analyzer [MCA]. The ADCAM module was interfaced to a laptop computer which ran an MCA emulator program to control the acquisition, and record the accumulated spectra.

The laboratory records that provide traceability of the sources used in the laboratory were used to specify the quantity of the isotopes on the certification date of the standard, and QUEST and SYNTH decayed the source to date of measurement. The detector manufacturers' quality assurance data sheets were used to specify the sodium iodide and germanium diode parameters. The absorbers specified were air, the detector end cap material, the iron casing surrounding the plutonium oxide source, and in the case of the germanium detectors the germanium dead layer. The source-to-detector distance and the sample time were specified, as were the system electronics settings for gain and zero. In the following examples, as parameterized in Table 4.1, the Compton continuum was remarkably free of deviation down to 500 keV, below which QUEST underestimated the experimental value. This underestimation is attributed to, (1) self-attenuation of the source, and (2) gamma-ray scattering in the

	HEU / Ge	Pu / Ge	Pu / NaI
Detector			
End Cap Thick. (mm)	0.51	0.51	0.51
Material	Al	Al	Al
Deadlayer Thick. (mm)	1.0	1.0	n/a
Diameter (cm)	6.0	6.1	12.7
Length (cm)	7.0	4.4	5.08
Efficiency (%)	40.0	30.04	n/a
Resolution	1.8 keV	2.1 keV	7.2%
Electronics			
Zero (keV)	0.0	0.0	0.0
Gain (keV/ch)	0.34	0.243	0.39
Linearity (keV/ch^2)	0.0	0.0	0.0
Channels	8192	4096	4096
Count Time (sec.)	34,669.08	1000	600
Sample Properties			
Area (cm^2)	1.0	17.18	17.18
Thickness (cm)	0.2	0.5	0.5
Mass (gm)	1.5	98.4	98.4
Distance (cm)	5.08	54.61	210.82
Bulk Matrix	100% Uranium	88% Pu / 12% O	88% Pu / 12% O
Absorber (cm)	n/a	0.6 Fe	0.6 Fe
Source Terms (gm)	^{235}U 1.40e+00 ^{238}U 1.00e-01 ^{232}U 2.00e-10	^{238}Pu 2.066e-02 ^{239}Pu 9.249e+01 ^{240}Pu 5.600e+00 ^{241}Pu 2.647e-01 ^{242}Pu 2.066e-02 ^{241}Am 5.510e-02	^{238}Pu 2.066e-02 ^{239}Pu 9.249e+01 ^{240}Pu 5.600e+00 ^{241}Pu 2.647e-01 ^{242}Pu 2.066e-02 ^{241}Am 5.510e-02
Decay Time	20 years	19.18 years	19.18 years

Table 4.1: Spectra comparison configuration.

lead shield of the detector, effects that are not currently simulated by QUEST's physics model.

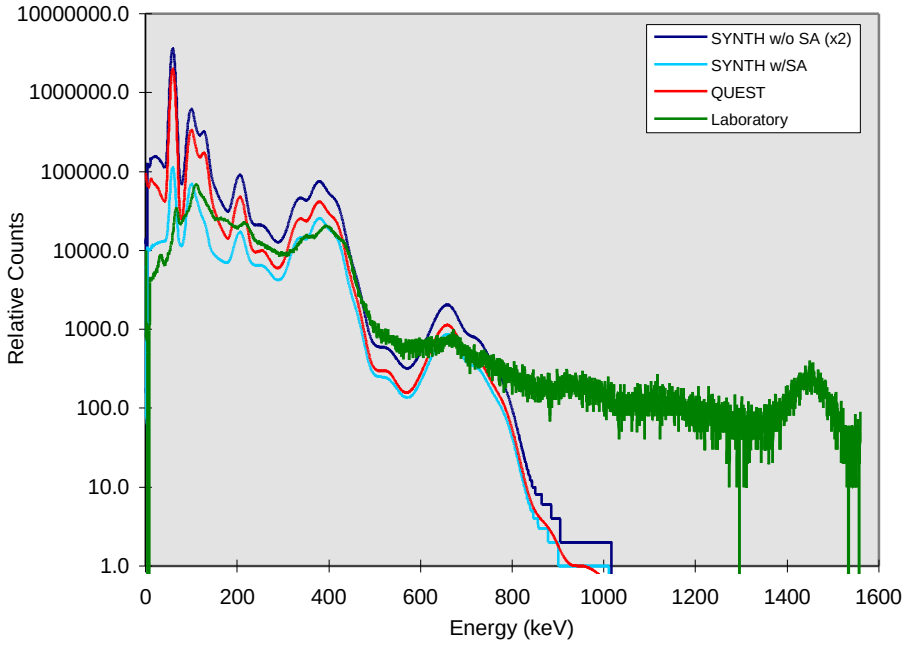


Figure 4.1: Comparison of Pu / NaI laboratory, synthetic SYNTH and QUEST spectra.

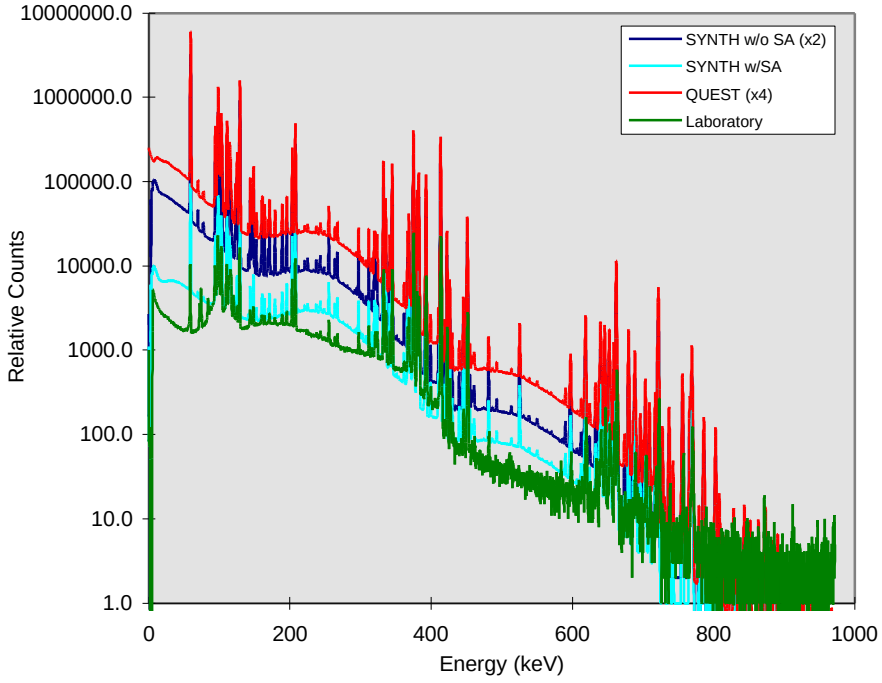


Figure 4.2: Comparison of Pu / Ge laboratory, synthetic SYNTH and QUEST spectra.

In Figure 4.1, QUEST and SYNTH were both setup to model the response of a 5"x2" NaI(Tl) detector to approximately 100 grams of a 20-year-old plutonium oxide source. As the laboratory spectrum was acquired with an unshielded detector assembly, it is composed of the signal from the Pu source as well as a significant contribution from the ambient background. In fact, almost all of the counts above 800 keV can be accounted for by the ^{208}Tl (a decay product of ^{232}Th), and ^{40}K that occur naturally in the soil and in concrete.

Below 800 keV, the shape of the laboratory spectrum is dominated by the signal from the Pu source and the Compton continuum from the high energy components of the ambient background. Although the energy calibrations were not fine tuned, both QUEST and SYNTH do reasonably well above 150 keV. Below that value, the differences between the two codes become more apparent. SYNTH generated spectra with and without self-attenuation are shown; SYNTH appears to overcorrect somewhat for self-attenuation effects in the sample. The QUEST code, on the other hand, uses the simpler point-source model (which makes no self-attenuation correction) and thus over predicts the activity of lower energy photons. In each spectra figure, legend indicated data series multipliers have been applied to ease comparison.

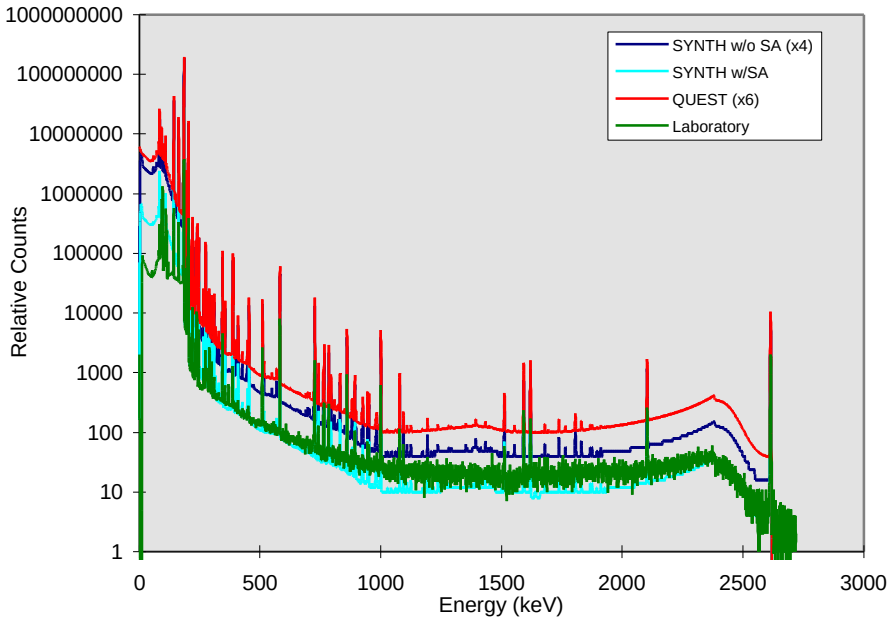


Figure 4.3: Comparison of HEU / Ge laboratory, SYNTH and QUEST synthetic spectra.

The same general characteristics may be seen in the plutonium spectra that were acquired with Ge detectors, shown in Figure 4.2. Other than the obvious difference in detector resolution, the laboratory Ge spectrum has a better signal-to-background ratio (the Ge detector was smaller and less efficient, so the source-to-detector distance was much smaller), and the energy calibration (keV/channel) is different by almost a factor of two.

In a separate experiment, a small quantity of Highly Enriched Uranium (HEU) was counted in the laboratory, and subsequently modeled by the SYNTH, and QUEST codes. The resulting spectra, shown in Figure 4.3, are dominated by the characteristic "signature" of ^{235}U (and its daughters) below 250 keV. Above that energy, the bulk of the activity is due to trace amounts (0.1 ppb) of ^{232}U in the material.

4.2 COMPARISON OF ALGORITHM RESPONSES

Data collection was done in a laboratory using standard radiological sources and off-the-shelf NaI gamma-ray detectors (Table 4.2). Detector output was collected during a "walk-by" inspection for a specific laboratory configuration (Figure 4.4). The data was fed to the QUEST detection algorithms and the algorithm output recorded. The data from the detectors was recorded at approximate one-second intervals, and consecutive samples were summed as necessary to simulate specific algorithms. Each inspection configuration was also simulated in QUEST using synthetic sources, detectors and paths, and the algorithm output was recorded and compared to that developed from the laboratory data.

The laboratory data acquisition system and software employed were originally developed by EG&G/Energy Measurements. Commercial off-the-shelf pre-amplifiers and amplifiers were utilized. The MCA employed for data acquisition was the EG&G ORTEC Model 920 Spectrum Master MCA. This MCA can acquire 1024-channel spectra from up to 16 detectors simultaneously. Each detector had its amplified output fed to separate input on the Model 920. Running multiple detectors into a single Model 920 had the advantage of reducing the number of separate MCAs required, but it did have a disadvantage of increasing the dead time. Real time is the actual wall clock time. Live time is the time during which the MCA is not busy processing a pulse and, hence, is available to accept a new pulse. Dead time is the difference between the real and live times, expressed as a percentage of the real time. Although individual spectra were taken for each detector, the dead time for each spectrum was the total MCA dead time (the sum of the dead time from all of the detectors connected to the 920), not just the dead time of that single detector. QUEST makes no provision for dead time in the sampling interval, so it was necessary to apply a correction factor to the algorithm inputs equal to the ratio of live time to real time.

Detector	**Bicron NaI** 3M3/3	**Harshaw NaI** 20MB/57-x 4
End Cap Thick. (mm)	0.5	0.5
Material	Al	Al
Diameter (cm)	7.62	12.7
Length (cm)	7.62	5.08
Resolution	7.4 %	9.8%
Electronics		
Zero (keV)	0.0	0.0
Gain (keV/ch)	3.0	3.0
Linearity (keV/ch^2)	0.0	0.0
Channels	1024	1024
Count Time (sec.)	1.0	1.0
Sample Properties		
Absorber (cm)	n/a	3.81 Pb, 0.32 Fe
Source Terms (gm)	^{238}Pu 2.10e-04 ^{239}Pu 9.395-e01 ^{240}Pu 5.688e-02 ^{241}Pu 2.690e-03 ^{242}Pu 2.10e-04 ^{241}Am 5.60e-04	^{60}Co 2.41e-03 mCi
Decay Time	26 years	n/a

Table 4.2: Algorithm comparison configuration.

In collecting the laboratory data, six paths were used following parallel lines approximately forty-two feet in length and six feet apart. The sources used were an unshielded, 2 cm diameter disk of ^{239}Pu, and a shielded ^{60}Co sample. Two different QUEST algorithms were used to evaluate the collected spectra: gross count and a windowed algorithm. The former computes the logarithm to the base two of the sum of the counts over all detector channels. The latter computes the logarithm to the base two of the difference between the counts in a window *A* (45 keV to 450 keV) and a window *B* (450 keV to 3 MeV) scaled by a factor *k*. *k* represents the ratio of the background counts in *A* and *B* for a reference background collected using the same detector. The output of both algorithms was normalized to zero.

Figures 4.5 and 4.6 show the algorithm output for the laboratory data and QUEST for the ^{60}Co sample and gross count algorithm. Here, the sampling period was one second, and the experimenter pushed the detector cart from left to right in front of the source. The graphs show in three-dimensions the relationship between inspector location (offset distance from the source in feet), position in time along the path (given in seconds), and the resulting algorithm response. Figure 4.7 shows a comparison of ^{239}Pu laboratory and QUEST windowed algorithm response. Here, only response values collected at offsets of six, twelve, and eighteen feet from the source are given. The sampling period was two seconds. The asymmetric shape of the laboratory data was a result of the shielding effects of the laboratory cart equipment layout and presence of the experimenter's body alongside the cart. In each of the experiments, QUEST slightly overestimates the algorithm response since it does not account for self-attenuation in the source.

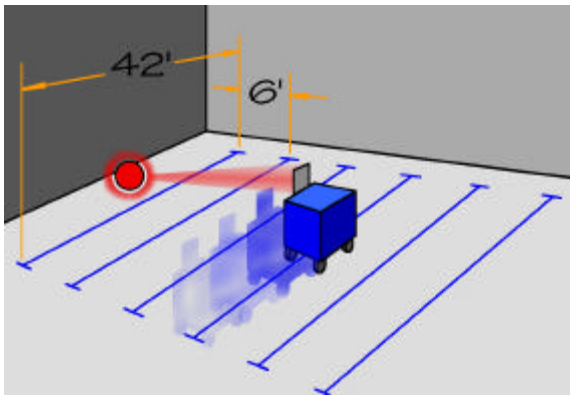


Figure 4.4: Laboratory algorithm response setup.

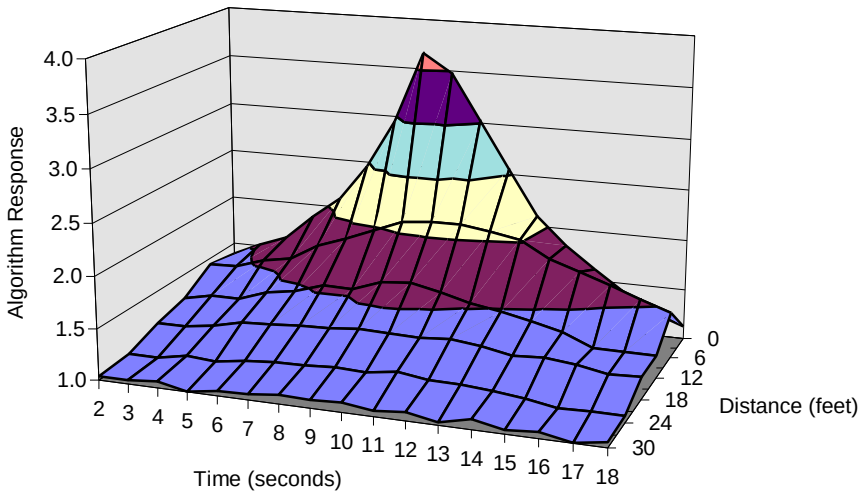


Figure 4.5: Laboratory ^{60}Co algorithm response.

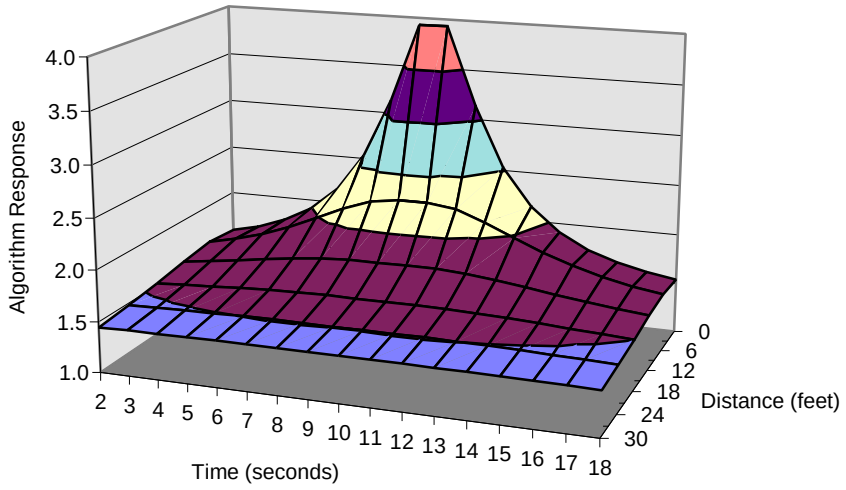


Figure 4.6: QUEST ^{60}Co algorithm response.

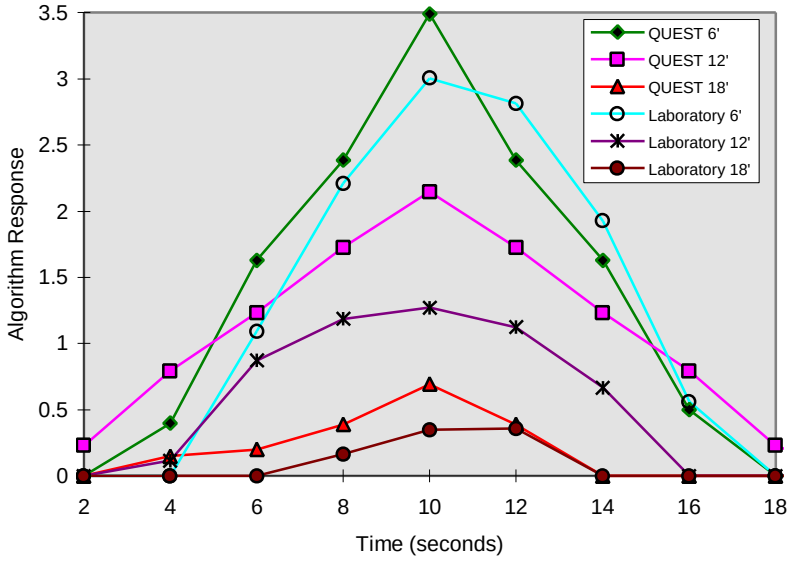


Figure 4.7: Comparison of ^{239}Pu laboratory and QUEST algorithm response.

5 SUMMARY

QUEST is proving to be a valuable tool that allows analysts, detector developers, and search managers to quantitatively explore the impact of technical or procedural changes on the nuclear material search process. The QUEST model provides a tool for examining the impact of new detector technologies, exploring alternative search concepts, studying new data fusion techniques, and a wide range of other practical studies.

QUEST was developed to be a portable, extensible simulation environment. Based in advanced object-oriented design methodologies, phenomenology is encoded in separate software modules that can be enhanced or replaced. Developed entirely in C++, OpenGL and World ToolKit, and the cross-platform user interface builder XVT, QUEST is portable between workstations (SGI Irix), and personal computer (Microsoft Windows) systems. In addition, the use of object file standards such as DXF and VRML makes possible the use of QUEST structures in other software packages.

Designed from the beginning to support large-scale, real-time, human-in-the-loop simulations of the nuclear material search process, QUEST provides timely and valuable scientific support to both search managers in the field, and analysts in the laboratory. Quantitative models such as QUEST allow searchers and inspection teams to optimize searches and maximize the probability of finding materials that can pose a threat. Presented experiments demonstrate the ability of QUEST to synthesize static source behavior and detector response. In addition, employing user configurable detection algorithms, QUEST gives realistic output from real-time, three-dimensional simulations. Using these capabilities, QUEST provides a means of calculating the probability of nuclear material detection within a given scenario.

Sandia National Laboratories remains dedicated to the support and development of QUEST. Work is underway to develop operational extensions to QUEST allowing for integration of both simulated and real-world searcher inputs. In addition, Sandia envisions a wide range of future applications for this simulation technology in areas as diverse as environmental monitoring, nuclear facilities inspections, and searcher training applications.

REFERENCES

A. V. Aho and J. D. Ullman, 1972. *The Theory of Parsing, Translation and Compiling, Vol. 1: Parsing*, Prentice-Hall, Englewood Cliffs, NJ.

Autodesk Inc., 1990. *AutoCAD Release 11 Reference Manual*, August 7, 1990.

G. Bell, A. Parisi, and M. Pesce, 1995. *The Virtual Reality Modeling Language: Version 1.0 Specification.*

A. D. Birrell and E. J. Nelson, 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 39-59.

Brick Institute of America, 1996. Personal communication through Brick Institute of America, 11490 Commerce Park Dr., Reston, VA 22091.

C. F. Delaney and E. C. Finch, 1992. *Radiation Detectors—Physical Principles and Applications*, Clarendon Press.

J. E. Dennis, Jr., and R. S. Schnabel, 1983. *Numerical methods for unconstrained optimization and nonlinear equations*, Prentice-Hall, Englewood Cliffs, NJ.

E. W. Dijkstra, 1968. Cooperating sequential processes. *Programming Languages*, F. Genuys (ed.), Academic Press, NY.

EG&G Ortec, 1991. *Detectors and Instruments Catalog*, EG&G Ortec, Oak Ridge, TN, 1991-1992.

G. Erdtmann and W. Soyka, 1979. *The Gamma Rays of the Radionuclides*, New York, Verlag Chemie.

G. Friedlander, J. W. Kennedy, E. S. Macias, and J. M. Miller, 1981. *Nuclear and Radiochemsitry*, 3rd Edition, New York, John Wiley & Sons.

R. Gunnink, 1977. An algorithm for fitting Lorentzian-broadened, K-series X-ray peaks of the heavy elements. *Nucl. Inst. and Methods*, V143, p. 145-149.

R. Gunnink and J. B. Niday, 1972. Computerized quantitative analysis by gamma ray spectrometry, Vol. 1: description of the GAMANAL program. *Technical Report UCRL-51061*, Lawrence Livermore National Laboratory, Livermore, CA.

R. Gunnink and A. L. Prindle, 1992. Nonconventional methods for accurately calibrating germanium detectors. *Journal of Radioanalytical and Nuclear Chemistry*, V160, pp. 305-314.

R. Gunnink, W. D. Ruhter, and J. B. Niday, 1988. GRPANAL: A suite of computer programs for analyzing complex Ge and alpha-particle detector spectra. *Technical Report UCRL-53861*, Vols. 1-3, Lawrence Livermore National Laboratory, Livermore, CA.

I. K. Helfer and K. M. Miller, 1988. Calibration factors for Ge detectors used for field spectrometry. *Health Physics*, Vol. 55, No. 1, pp. 15-29, July, 1988.

W. Hensley, A. McKinnon, H. Miley, M. Panisko, and R. Savard, 1994. SYNTH: A spectrum synthesizer. In *Proceedings of the 35th Annual Meeting of the INMM*, pg. 629.

International Conference of Building Officials, 1994. *1994 Uniform Building Code*, Vol. 3, International Conference of Building Officials, p. 603.

Institute of Electrical and Electronics Engineers, 1993. *IEEE Standard Multichannel Analyzer (MCA) Histogram Data Interchange Format for Nuclear Spectroscopy*, IEEE Std. 1214-1992, New York, NY, January 28, 1993.

Internet Engineering Task Force, 1994 (RPC). *Internet-Draft*: *RPC: Remote Procedure Call Protocol Specification Version 2*, Internet Engineering Task Force, Corporation for National Research Initiatives, Reston, VA, May 31, 1994, (http://www.ietf.cnri.reston.va.us).

Internet Engineering Task Force, 1994 (XDR). *Internet-Draft: XDR: External Data Representation Standard*, , Internet Engineering Task Force, Corporation for National Research Initiatives, Reston, VA, May 31, 1994 (http://www.ietf.cnri.reston.va.us).

M. M. Johnson, M. E. Goldsby, T. D. Plantenga, W. B. Wilcox, and W. K. Hensley, 1996. A model to quantify uncertain emergency search techniques, theory and application. In *Proceedings of the 37th Annual Meeting of the Institute of Nuclear Materials Management (INMM)*, Naples, Florida.

M. M. Johnson, M. E. Goldsby, W. B. Wilcox, W. K. Hensley, and R. G. Hansen, 1997. Computational models to quantify uncertain emergency search techniques—a comparison of measured and synthetic gamma-ray detector response functions. In *Proceedings of the 38th Annual Meeting of the Institute of Nuclear Materials Management (INMM)*, Phoenix, Arizona.

J. Kopecky, W. Ratynski, and F. Warming, 1967. Curves for the response of a Ge (Li) detector to gamma rays up to 11 MeV. *Nucl. Inst. and Methods*, V50, p. 333.

K. M. Miller and P. Shebell, 1993. In situ gamma-ray spectrometry: a tutorial for environmental radiation scientists. Environmental Measurements Laboratory, U.S. Department of Energy, New York, NY, October, 1993.

R. Packard, 1981. *Architectural Graphic Standards–7th Edition*, John Wiley and Sons, Inc.

rpcgen Manual Page. View the manual pages for *rpcgen* by typing *man rpcgen* at the command line of any of the following UNIX-variant operating systems: SunOS (Sun Microsystems), Solaris (Sun Microsystems), IRIX (Silicon Graphics), OSF (Digital Equipment Corporation, among others), Linux (copylefted freeware), or see the online manual pages at http://www.uwaterloo.ca/man.

Sense 8 Corporation, 1995. *World ToolKit Version 2.1 Reference Manual*, Sense8 Corporation, Mill Valley, CA.

U.S. Department of Health, Education, and Welfare, 1970. *Radiological Health Handbook*, U.S. Department of Health, Education, and Welfare, Washington, DC.

R. T. Weidner and R. L. Sells,1973. *Elementary Modern Physics*, 2nd edition, Allyn & Bacon, Boston.

J. Wernecke, 1994. *The Inventor Mentor*, Addison-Wesley, Reading, MA.

XVT Software Inc., 1994. *XVT-Portability Toolkit Reference*, XVT Release 4.0, XVT Software Inc., December,1994.

BIBLIOGRAPHY

Detector Physics and Gamma-Ray Scattering

Aic Software, *Photocoef: A Nuclear Physics Utilities Program Written for IBM PCs*, Aic Software, Grafton, MA, January 1995.

Attix, F. H., and Roesch, W. C., *Radiation Dosimetry*, 2nd edition, Academic Press, New York, 1968.

Berger, R. T., "The x- or gamma-ray absorption or transfer coefficient: tabulations and discussion", *Radiation Research*, v15, 1-29, 1961.

Bergey, J. A., and Scott, H. L., *System Effectiveness Model (SEM)*, software v1.03, Sandia National Laboratories, January 6, 1994.

Cohen, D. D., and Clayton, E., "Ion induced x-ray emission", Chapter 5, *Ion Beams for Materials Analysis*, Academic Press, Australia, 1989.

Colbert, H. M., "SANDYL—A computer program for calculating combined photon-electron transport in complex systems", *Technical Report SLL-74-0012*, Sandia National Laboratories, May 1974.

Crouthamel, C. E. (ed.), *Applied Gamma-Ray Spectrometry*, Pergamon Press Inc., New York, 1960.

Davisson, C. M., and Evans, R. D., "Gamma-ray absorption coefficients", *Reviews of Modern Physics*, v24, n2, April 1952, pp. 79-107.

Debertin, K. and Helmer, R. G., *Gamma- and X-Ray Spectrometry with Semiconductor Detectors*, North-Holand, 1988.

Dowdy, E. J., Henry, C. N., Hastings, R. D., and France, S. W., "Neutron detector suitcase for the nuclear emergency search team", *Technical Report LA-7108*, Los Alamos Scientific Laboratory, New Mexico, Februray 1978.

Duderstadt, J. J., and Martin, W. R., *Transport Theory*, John Wiley & Sons, New York, 1979.

Eisberg, R. M., *Fundamentals of Modern Physics*, John Wiley & Sons, New York, 1961.

Flugge, S., *Practical Quantum Mechanics*, Springer-Verlag, Berlin, 1974.

Glascock, M. D., "Tables for neutron activation analysis", technical report, University of Missouri, Research Reactor Facility, March 1991.

Glasstone, S., and Sesonske, A., *Nuclear Reactor Engineering*, Van Nostrand Reinhold Company, New York, 1981.

Grodstein, G. W., *X-ray Attenuation Coefficients From 10 kev to 100 Mev*, United States Department of Commerce, National Bureau of Standards Circular 583, April 30, 1957.

Gunnink, R., and Niday, J. B., "Computerized quantitative analysis by gamma ray spectrometry, Vol. 2: source listing of the GAMANAL program", *Technical Report UCRL-51061V2*, Lawrence Livermore National Laboratory, Livermore, California, December 6, 1971.

Gunnink, R., and Niday, J. B., "Computerized quantitative analysis by gamma ray spectrometry, Vol. 3: a user's guide to GAMANAL", *Technical Report UCRL-51061V3*, Lawrence Livermore National Laboratory, Livermore, California, July 8, 1971.

Gunnink, R., and Niday, J. B., "Computerized quantitative analysis by gamma ray spectrometry, Vol. 4: auxiliary programs for GAMANAL", *Technical Report UCRL-51061V4*, Lawrence Livermore National Laboratory, Livermore, California, June 1, 1972.

Heitler, W., *The Quantum Theory of Radiation*, 3rd edition, Oxford at the Clarendon Press, London, 1954.

Hubbell, J. H., *Photon Cross Sections, Attenuation Coefficients, and Energy Absorption Coefficients From 10 keV to 100 GeV*, United States Department of Commerce, National Bureau of Standards, NSRDS-NBS 29, August 1969.

Jackson, J. D., *Classical Electrodynamics*, John Wiley & Sons, 1975.

Jaeger, R. G., Blizard, E. P., Chilton, A. B., Grotenhuis, M., Honig, A., Jaeger, Th. A., and Eisenlohr, H. H. (ed.), *Engineering Compendium on Radiation Shielding, Volume I: Shielding Fundamentals and Methods*, Springer-Verlag, New York, 1968.

Jaeger, R. G., Blizard, E. P., Chilton, A. B., Grotenhuis, M., Honig, A., Jaeger, Th. A., and Eisenlohr, H. H. (ed.), *Engineering Compendium on Radiation Shielding, Volume II: Shielding Materials*, Springer-Verlag, New York, 1975.

Jauch, J. M. and Rohrlich, F., *The Theory of Photons and Electrons—The Relativistic Quantum Field Theory of Charged Particles with Spin One-half*, Springer-Verlag, 1976.

Jenkins, T. M., Nelson, W. R., and Rindi, A. (ed.), *Monte Carlo Transport of Electrons and Protons*, Plenum Press, New York, 1987.

Johns, H. E., Cormack, D. V., Denesuk, S. A., and Whitmore, G. F., "Initial distribution of Compton electrons", *Canadian Journal of Physics*, v30, 1952, pp. 556-564.

Meyerhof, W. E., *Elements of Nuclear Physics*, McGraw-Hill Inc., 1967.

Mitchell, D. J., "GADRAS-PC1, Gamma detector response and analysis software", *Technical Report SAND92-0285*, Sandia National Laboratories, May 1992.

Mitchell, D. J., Laub, T. W., and Marlow, K. W., "Semi-empirical response function for neutron detectors", *Technical Report SAND93-2570*, Sandia National Laboratories, October 1993.

Moss, C. E., Byrd, R. C., Feldman, W. C., Auchampaugh, G. F., Estes, G. P., Ewing, R. I., and Marlow, K. W., "The detection of uranium-based nuclear weapons using neutron-induced fission", *Technical Report LA-UR-91-3514*, Los Alamos National Laboratories, December 1991.

Nelms, A. T., *Graphs of the Compton Energy-Angle Relationship and the Klein-Nishina Formula from 10 Kev to 500 Mev*, United States Department of Commerce, National Bureau of Standards Circular 542, August 28, 1953.

Nelson, W. R., "The EGS4 monte carlo simulation of the coupled transport of electronics and photons", Oak Ridge National Laboratory, RSIC Computer Code Library, CCC-331, Oak Ridge, TN, contributed by Stanford Linear Accelerator Center, Stanford University, September 1994.

O'Dell, R. D., and Alcouffe, R. E., "Transport calculations for nuclear analysis: theory and guidelines for effective use of transport codes", *Technical Report LA-10983-MS*, Los Alamos National Laboratory, September 1987.

Price, B. T., Horton, C. C., and Spinney, K. T., *Radiation Shielding*, Pergamon Press, New York, 1957.

Quittner, P., *Gamma-Ray Spectroscopy*, Halsted Press, New York, 1972.

Rice, A. F., and Roussin, R. W. (ed.), "Deterministic methods in radiation transport, a compilation of papers presented February 4-5, 1992", *Technical Report ORNL/RSIC-54*, Oak Ridge National Laboratory, Radiation Shielding Information Center, June 1992.

Roy, R. R., and Reed, R. D., *Interactions of Photons and Leptons with Matter*, Academic Press, New York, 1968.

Segre, E. (ed.), Staub, H., Bethe, H., Ashkin, J., Ramsey, N. F., and Brainbridge, K. T., *Experimental Nuclear Physics*, John Wiley & Sons, New York, 1953.

Strode, J. N., and Van Tuyl, H. H., "PUSHLD—A code for calculation of gamma dose rates from plutonium in various geometries", *Technical Report HEDL-TME 73-89*, Hanford Engineering Development Library.

Tait, W. H., *Radiation Detection*, Butterworth & Co Ltd, Boston, 1980.

Structure and Material Sources

AutoDesk Inc., *AutoCAD LT User's Guide, AutoCAD LT Release 2 for Windows*, Autodesk Inc., April 12, 1995.

ASM International, *Engineered Materials Handbook–Vol. 4–Ceramics and Glasses*, ASM International, 1991.

Bansal, N. and Doremus, R., *Handbook of Glass Properties*, Academic Press, Inc., 1986.

Bowes, W., Russell, L., Suter, G., *Mechanics of Engineering Materials*, John Wiley and Sons, 1984.

Brady, G. and Clauser, H., *Materials Handbook 13th Edition*, McGraw Hill, 1991.

Carmichael, R., *CRC Practical Handbook of Physical Properties of Rocks and Minerals*, CRC Press, Inc., 1989.

Hornbostel, C., *Construction Materials–Types, Uses, and Applications*, John Wiley and Sons, Inc., 1991.

Illston, J., Dinwoodie, J., and Smith, A., *Concrete, Timber, and Metals–The Nature and Behavior of Structural Materials*, Van Nostrand Reinhold, 1979.

Jastrzebski, Z., *The Nature and Properties of Engineering Materials*, 2nd edition, John Wiley and Sons, 1977.

Lubin, G., *Handbook of Fiberglass and Advanced Plastics Composites*, Van Nostrand Reinhold, 1969.

McGuinness, W. and Stein, B., *Building Technology– Mechanical and Electrical Systems*, John Wiley and Sons, 1977.

Packard, R., *Architectural Graphic Standards–7th Edition*, John Wiley and Sons, Inc., 1981.

Schwartz, C. and Turner, R., *Encyclopedia of Associations*, Gale Research Inc., 1995.

Shackelford, J., *Introduction to Materials Science for Engineers*, Macmillan Publishing Company, 1985.

Young, J. and Shane, R., *Materials and Processes*, 3rd edition, Marcel Dekker, Inc., 1985.

Programming and Interprocess Communication

Andrews, M., *C++ Windows NT Programming*, M&T Books, New York, NT, 1994.

Athas, W. C., and Seitz, C. L., "Multicomputers: message-passing concurrent computers", *IEEE Computer*, Aug. 1988, pp. 9-23.

Bach, M. J., *The Design of the UNIX Operation System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

Erikson, C., "Error correction of a large architectural model: the henderson county courthouse", *Technical Report TR95-013*, Department of Computer Science, UNC-Chapel Hill, 1995.

Garcia-Alonso, A., Serrano, N., and Flaquer, J., "Solving the collision detection problem", *IEEE Computer Graphics and Applications*, May 1994.

Institute of Electrical and Electronics Engineers, *POSIX¾ System Application API¾ Threads & Extensions*, Project Number 1003.1c*, Draft Version D10, Sept., 1994 (http://stdsbbs.ieee.org/products/catalog).

Iona Technologies, Ltd., *Orbix Programming Guide*, Iona Technologies, Ltd., Dublin, Ireland, 1995.

B. Janssen, D. Severson, and M. Spreitzer, 1995. *ILU 1.7 Reference Manual*, XeroxCorporation, Palo Alto, CA, (ftp://ftp.parc.xerox.com/pub/ilu/ilu.html).

Keppel, D., "Tools and techniques for building fast portable threads packages", *Technical Report UWCSE 93-05-06*, University of Washington, Seattle, WA (ftp://ftp.cs.washington.edu/tr).

Laidlaw, D. H., and Hughes, J. F., "Constructive solid geometry for polyhedral objects", *Siggraph*, v20, n4, 1986.

Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 1.1, Object Management Group, 1992 (http://www.omg.org).

Requicha, A., “Representations for rigid solids: theory, methods, and systems”, *Computing Surveys*, v12, n4, December 1980.

Schmidt, D. C., Harrison, T., and Al-Shaer, E., “Object-oriented components for high-speed network programming”, In *USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June, 1995.

Thibault, W. C., and Naylor, B. F., “Set operations on polyhedra using binary space partitioning trees”, *Computer Graphics*, v21, n4, July 1987.

User Interface Development

XVT Software, Inc, *Guide to XVT Development Solution for C*, XVT Software Inc., December1994.

XVT Software, Inc, *Guide to XVT Development Solution for C++*, XVT Software Inc., November1994.

XVT Software, Inc, *XVT-Architect Manual*, XVT Software Inc., XVT-Architect 1.0, March 1995.

XVT Software, Inc, *XVT Graphical Extensions*, Cygnus Engineering, February 1995.

XVT Software, Inc, *XVT Graphical Extensions Interface Document*, Cygnus Engineering, February 1995.

XVT Software, Inc, *XVT Platform-Specific Book for Windows NT*, XVT Software Inc., December1994.

XVT Software, Inc, *XVT-Power++ Reference*, XVT Software Inc., November1994.

APPENDIX A: SOFTWARE INSTALLATION

QUEST v2.01 is a Microsoft Win32 (Windows 95/98, Windows NT v3.51/4.0/5.0) application. The following lists both minimum and recommended system requirements for running the PC version of QUEST.

Minimum Requirements	Recommended System
Intel Pentium 100 MHz system with Microsoft Windows.	Dual Intel Pentium II (266 MHz) system with Microsoft Windows NT v4.0.
24 MBytes of system memory.	36 MBytes of system memory.
VGA Video controller and monitor (640x480 resolution).	OpenGL accelerated video card (e.g. Integraph) and monitor (800x600 resultion).
Hard drive with ~50 MBytes free space.	Hard drive with 100 MBytes free space.

QUEST is distributed as a single PC zip archive. To install the application, create a clean parent directory on the target machine (e.g. "`c:\Quest201`"), and extract the archive into that directory—remembering to include the "–d" archive switch to create subdirectories. The application is divided into a number of directories. Prior to starting QUEST the host computer must have ODBC libraries installed. To determine if the ODBC drivers are already on the target machine, check the Windows Control Panel. If the ODBC manager is not present, the appropriate drivers can be installed by executing ODBC setup, "`<parent>\Program\Odbc\setup.exe`". In addition, QUEST requires the presence of the graphics display library OpenGL. OpenGL is an integral part of Microsoft Windows NT, and hence NT users can use QUEST directly. Windows 95 users must move the OpenGL software library from "`<parent>\Program\Libraries\Opengl32.dll`" to the application program directory, "`<parent>\ Program\`". Finally, the QUEST application is started by selecting the primary executable "`<parent>\Program \QUEST.exe`".

QUEST enforces an application directory structure, as well as the consistent use of case insensitive names and the ISO 8.3 compliant file naming convention. Assume QUEST will be stored in some parent directory, the name of which is unimportant, call it "`Quest`". Then the QUEST directory structure will be maintained beneath "`Quest`" as follows:

`Quest\`	Parent directory (name unimportant).
`Quest\Bkground`	Background objects (`*.bkg`).
`Quest\Database`	Read-only databases used by the TP and GUI.
`Quest\Detector`	Detectors objects (`*.dtc`).
`Quest\Dxfs`	Original AutoCAD structure input files (`*.dxf`).
`Quest\Material`	Material databases (`*.mdb`).
`Quest\Path`	Active object paths (`*.pth`, `*.wtk`).
`Quest\Program`	QUEST application program files.
`Quest\Program\Libraries`	Additional program libraries required on certain platforms.
`Quest\Program\Odbc`	Microsoft ODBC installation files.
`Quest\Reports`	Generated output reports (`*.txt`, etc.).
`Quest\Scenario`	Scenario configuration files (`*.scn`).
`Quest\Source`	Sources objects (`*.src`).
`Quest\Spectrum`	Run-time output files (`*.dat`, `*.otp`).
`Quest\Struct`	Structure input files (`*.qsf`, `*.str`).

It should be noted that QUEST is computationally intensive, and will benefit by being run on a multiprocessor PC, and systems with an OpenGL graphics accelerator.

Structure Files

QUEST includes a number of structures, both in DXF and QUEST specific QSF formats for use in search simulations. The following listing details the name of each structure, its description, and the structure's relative graphics complexity specified in number of polygons.

Name	Description	#Polygons
apart1.str	Apartment: first of four structures that make up a multi-story apartment building.	128
apart2.str	Apartment: second of four structures that make up a multi-story apartment building	128
apart3.str	Apartment: third of four structures that make up a multi-story apartment building	54
apart4.str	Apartment: fourth of four structures that make up a multi-story apartment building	1
apartmnt.str	Apartment: multi-story apartment building made up of structures apart1, apart2, apart3 and apart4.	311
base920.str	First floor of a government-style building without a roof.	248
roof920	Roof for government-style building. Must be translated into place.	1
gov_1.str	Simple one-story government-style building with a roof made up of structures base920 and roof920.	249
gov_2.str	Two-story government-style building with two instances of structure base920 and one roof920.	497
hosptl1.str	Hospital: first of four structures that make up a multi-story hospital building.	1902
hosptl2.str	Hospital: second of four structures that make up a multi-story hospital building	617
hosptl3.str	Hospital: third of four structures that make up a multi-story hospital building	596
hosptl4.str	Hospital: fourth of four structures that make up a multi-story hospital building	8
hospital.str	Hospital: multi-story hospital building made up of structures hosptl1, hosptl2, hosptl3 and hosptl4.	3123
rhouse1.str	RHouse: first of three structures that make up a multi-story roofed house.	998
rhouse2.str	RHouse: second of three structures that make up a multi-story roofed house.	72
rhouse3.str	RHouse: third of three structures that make up a multi-story roofed house.	4
rhouse.str	RHouse: multi-story roofed house made up of structures rhouse1, rhouse2 and rhouse3.	1074
office1.str	Office: first of three structures that make up a multi-story office building.	468
office2.str	Office: second of three structures that make up a multi-story office building.	465
office3.str	Office: third of three structures that make up a multi-story office building.	1
office.str	Office: multi-story office building made up of structures office1, office2 and office3.	934
printer1.str	Printer Building: first of three structures that make up a multi-story printer building (office building).	1478
printer2.str	Printer Building: second of three structures that make up a multi-story printer building.	1565
printer3.str	Printer Building: third of three structures that make up a multi-story printer building (office building).	1
printer.str	Printer Building: muli-story printer building composed of printer1, printer2 and printer3.	3044
shouse1.str	Simple House: first of three structures that make up a simple multi-story house.	372
shouse2.str	Simple House: second of three structures that make up a simple multi-story house.	414
shouse3.str	Simple House: third of three structures that make up a simple multi-story house.	1
shouse.str	Simple House: simple multi-story house made up of structures shouse1, shouse2 and shouse3.	787
wall1.str	Wall1: single polygon representing a wall with a grid on both sides of the wall (used for testing).	33
cube.str	Cube: simple cube structure (used for testing).	6

APPENDIX B: STRUCTURE CREATION

QUEST supports the input and manipulation of structure files specified in the three-dimensional (3D) AutoCAD DXF format. While a complete treatment of structure file creation using a Computer Aided Design (CAD) package is beyond the scope of this paper, this section details the creation of a simple example structure for use with QUEST. This example details use of Autodesk's AutoCAD LT v2.0 for Microsoft Windows 95/NT. However, QUEST input structures can be created with any CAD package supporting the 3D DXF file format. The completed drawing need only be saved as a Drawing Interchange File format (DXF) file; QUEST will only accept drawings made up of 3DPOLYs and 3D faces.

The DXF file format is an ASCII-character based file format that can be imported into and rendered by a wide variety of 3D graphics rendering programs. The data in DXF files can also be easily manipulated in application-specific ways. QUEST converts the drawing data in the DXF files to a format that is more suited to fast data access. These converted files are called QUEST Structure Files (QSF). The user need not worry about creating these QSF files as they are created by a conversion program within QUEST.

QUEST's 3D graphics interface is created through the use of the graphics programming library World ToolKit (WTK). WTK will only render "closed" 3D solid-fill polygons. And, by definition, a "closed" polygon is one that contains at least three unique points, the first and last of which must be connected by a line segment to close the polygon. All other AutoCAD drawing commands, including 2D lines (e.g. the LINE command in AutoCAD) are ignored and not graphically rendered by QUEST.

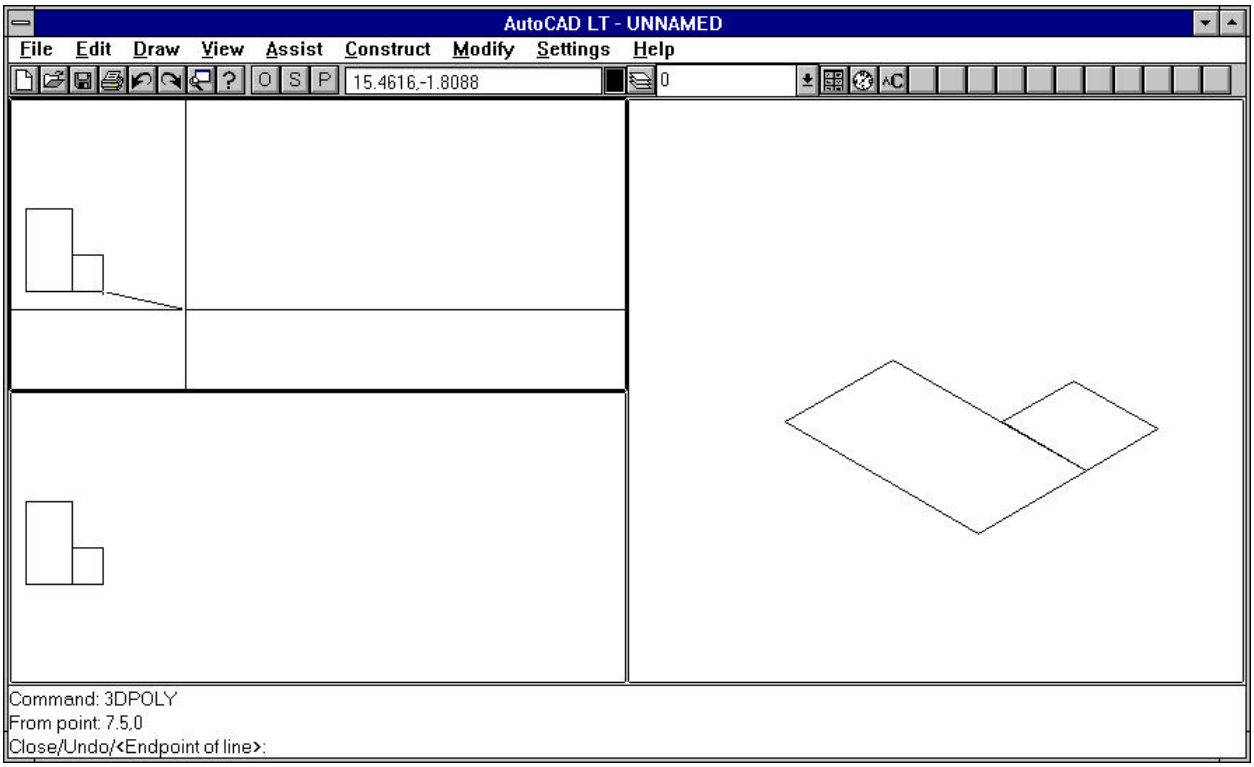


Figure B.1: Creating 3DPOLYs.

B.1 CREATING A DXF FOR USE IN QUEST

Creating structural drawings for QUEST is not difficult, but time consuming. We recommend that the person creating the drawings have at least a minimal working knowledge of AutoCAD and CAD drawing methods (refer to the AutoCAD User's Guide and tutorials).

The best way to start a new DXF file is to setup a prototype file. In the prototype setup the following: Grid Lines, Standard Units, Viewports, Layers, and Blocks. It should be noted that the command for 3DPOLY is not in a pull down menu selection and must be entered at the command line (see Figure B.1). In AutoCAD, the user will only see the outline of each polygon. When the polygons are rendered in QUEST's 3D graphics the outlined polygons will be rendered as solid-filled polygons.

Grid lines in AutoCAD assist the user in creating a drawing. They are not used in QUEST nor do they have any effect on the structural drawing (see "grid" in AutoCAD User's Guide). The recommended unit to use in creating an AutoCAD structural drawing is "DECIMAL" (see UNITS command in AutoCAD User's Guide). The recommended measure should be ENGLISH (see Setting the System of Measurement in AutoCAD User's Guide) (Figure B.2). For example, 10.3 is ten feet and three tenths of a foot. It is not 10 feet and 3 inches.

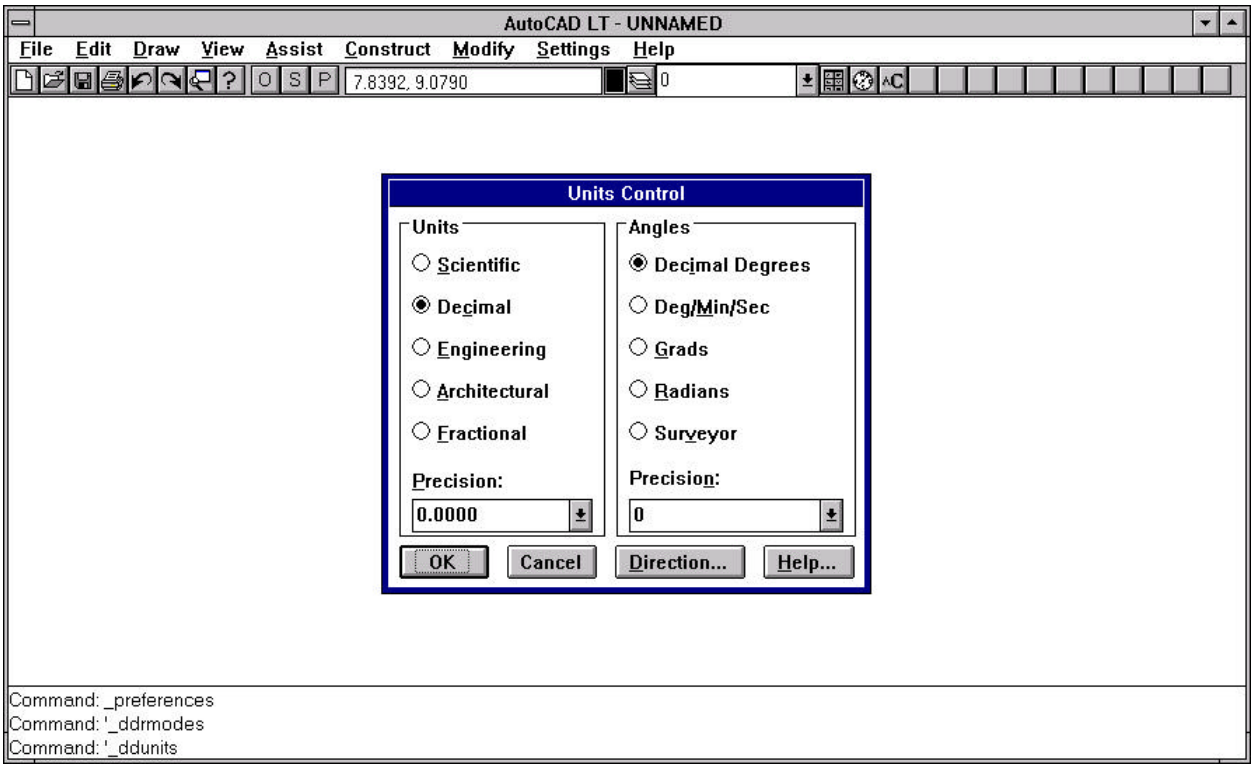


Figure B.2: Setting drawing units.

Multiple *viewports* may be defined in AutoCAD. Viewports allow the user to set different viewing angles and different scales for each viewport. This will allow the user to see the drawing from an arbitrary prespective. Use layers to define different groups of objects (Figure B.3). For example, doors may be defined in one layer, on another layer interior walls may be defined. The user may assign different colors to each layer. By using layers and making some layers visible and others invisible, different levels of detail of the structure may be seen within the AutoCAD drawing. The color associated with the layer will be the color rendered in QUEST for the graphical objects drawn on that layer.

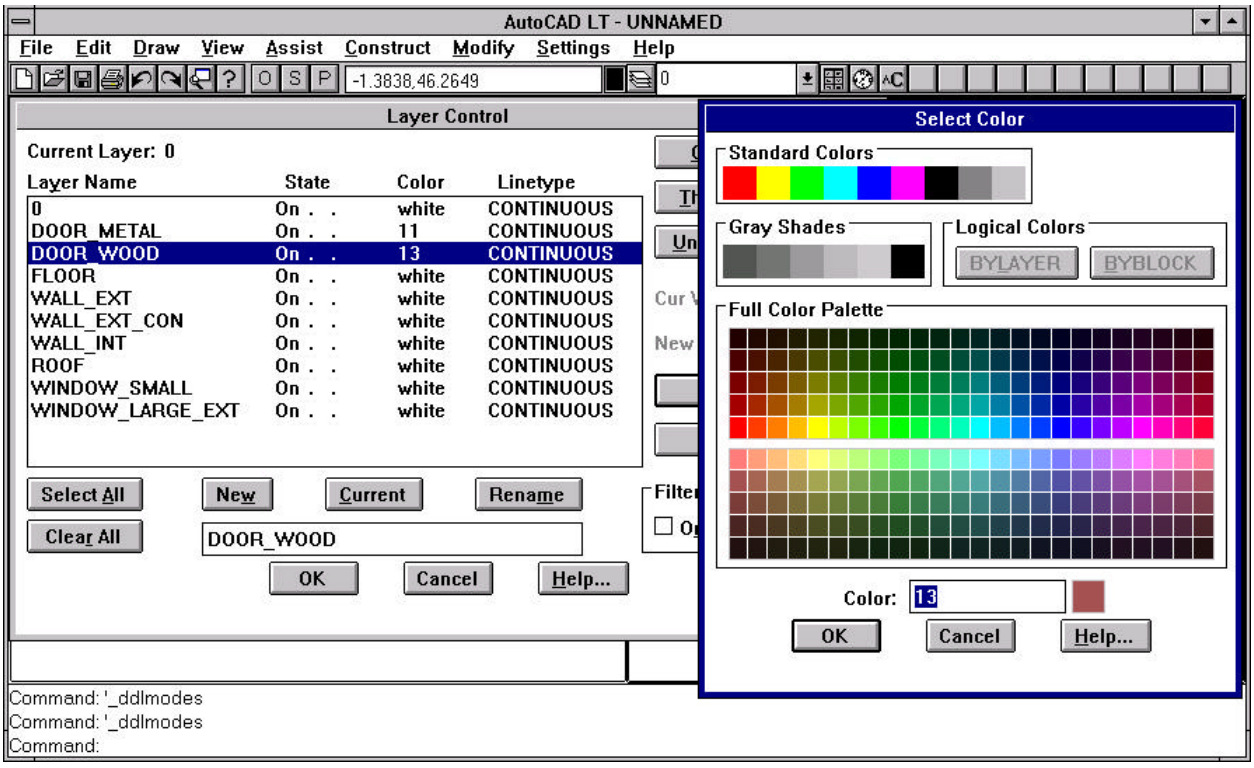


Figure B.3: Defining layers.

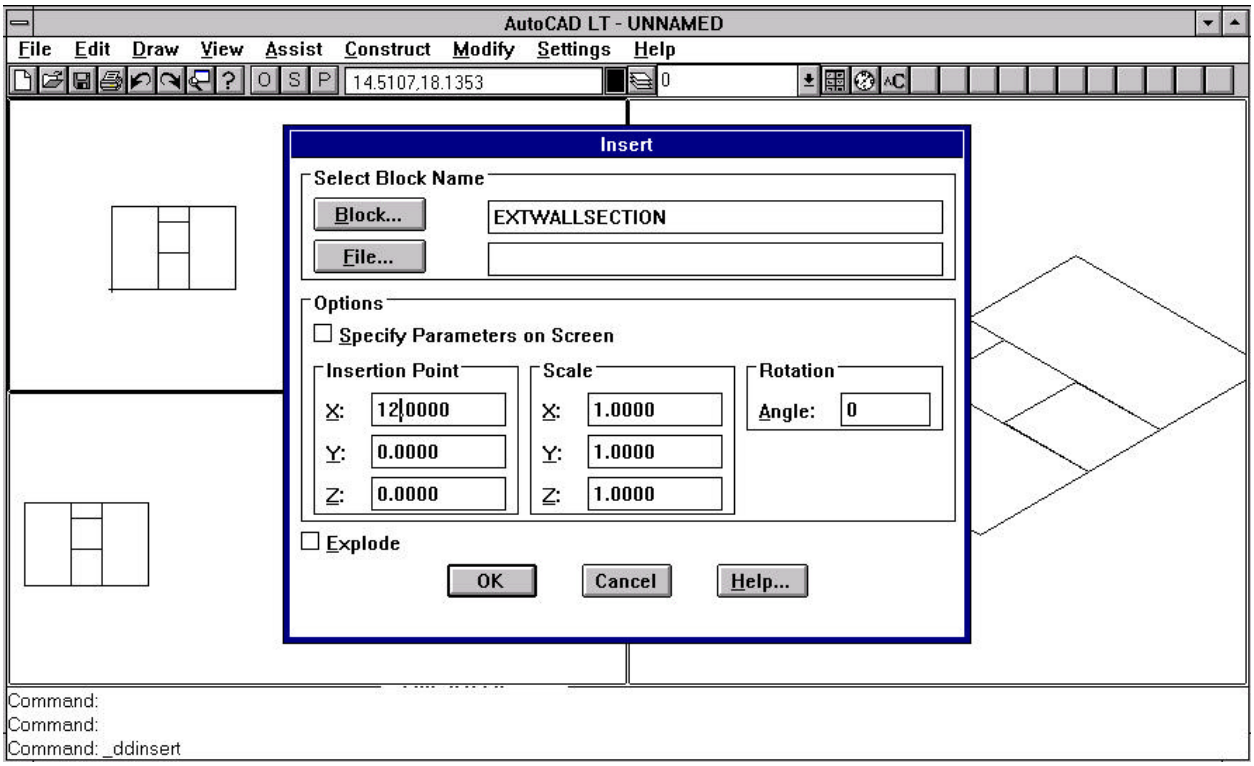


Figure B.4: Creating blocks.

Layers should be given names that associate it with the objects drawn on those layers (see "layer" in AutoCAD User's Guide). Examples of layer names include: DOOR_WOOD, for wooden doors; DOOR_METAL, for metal doors; WALL_INT, for interior walls; FLOOR, for floors; and ROOF, for the roof.

A *BLOCK* is a group of graphical entities (Figure B.4). In some buildings there may be many wall sections that are exactly alike. Instead of drawing each part of a wall section repeatedly, a BLOCK can be created to represent one section of the wall and inserted in different locations (translated and/or rotated) to create the building. BLOCKs may be inserted into other BLOCKs. Elementary components, which are individual polygons, and BLOCKs form the hierarchy of structural picking within QUEST for material assignments.

For convenience sake, the front left bottom corner of the drawing of the building should be assigned the origin of the building ($X = 0.0$, $Y = 0.0$, $Z = 0.0$). The building can later be positioned in any location within QUEST's structure layout. The drawing of the building must be laid out such that when viewing the building from the outside looking at what would be the front of the building, AutoCAD's Universal Coordinate System positive X axis is pointing to the right, the positive Y axis is pointing in the viewing direction away from the front of the building towards the back of the building and the positive Z axis is up (Figure B.5).

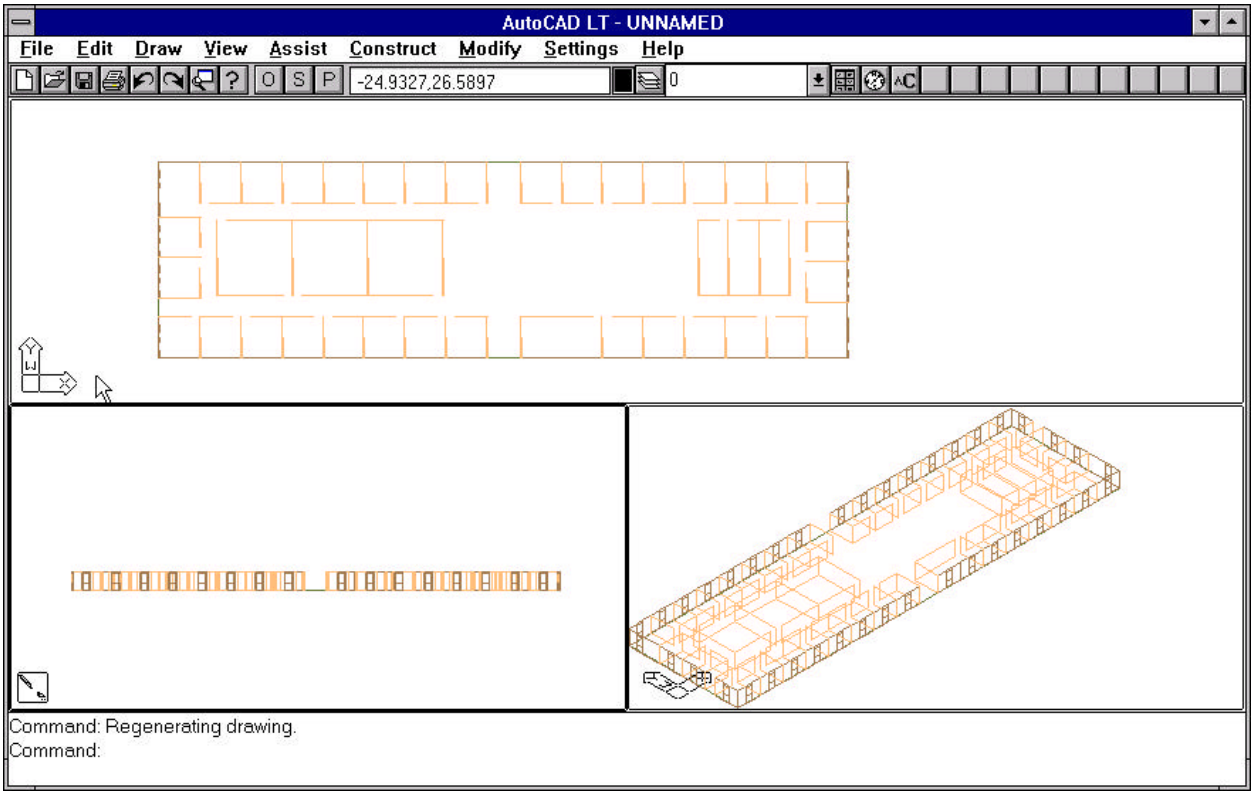


Figure B.5: Setting coordinate axis.

B.2 Drawing Detail and Levels

The drawing of a structure should be made as simple as possible. Walls should be represented as single planes (no thickness). Try to avoid making walls with real thickness (creating cubes). Wall thickness is represented in QUEST by material type and thickness assignments the user may set. Try to avoid adding "decorator" type features to the structural drawings. Because the 3D graphics of QUEST is dynamic, a large number of unnecessary polygons may cause QUEST's 3D graphics rendering to appear slow or "choppy".

When creating multi-floor structural drawings, attempt to create a single DXF file for each floor of the building. QUEST's 3D graphics favor buildings built in "levels". A level should consist of a single floor of a building and all interior and exterior walls that make up that level of the building. Do not make a separate ceiling for the current level. The floor of the level above should represent the ceiling for the current level. Make a separate level for roofs. Building structural drawings in "levels" allows QUEST to remove levels or floors above the current level so the user may have a top-down view into the current level. It also prevents QUEST from needing to render the graphic of levels that are not currently visible. This floor culling allows QUEST to render the graphic scenes more quickly. If a building is made up of floors that are identical, the user may create one level in AutoCAD and insert and translate that one copy of the level to create a multi-floor building inside the QUEST application.

When the structural drawing is completed or the user wants to quit AutoCAD, use the "File" pull down menu and select "Save As". In the Save File As dialog box select "DXF (`*.DXF`)" in the type menu selection. Give the file a name and select OK. When the DXF file is to be opened for editing, use the "File" pull down menu and select "Open". In the "Open File" dialog box choose "DXF (`*.DXF`)" in the List Files of Type menu selection and the list of DXF file will be made visible. Choose the file desired and select OK.

APPENDIX C: RAW BUILDING MATERIALS

Type	Breakdown	Description	Chemical Makeup	Density (gm/cm^3)
Background	Nothing			0
	Air		N 75.5 O 23.2 A 1.3	0.0013
Sand, Soil,				
and Gravel				
	Sand, dry		Si 46.7 O 53.3	1.3
	Brick	common, burned (Average)	O 48.0 Si 30.4 Al 10.1 Fe 3.7 K 2.2 Mg 1.5 Ca 1.4 Na 0.7 S 0.3	1.8-2.0 (avg. 1.96)
	Gravel	½ granite and ½ limestone. For particular gravels, see Stone	O 48.0 Si 18.4 Ca 17.5 C 5.2 Al 4.2 K 2.5 Fe 1.8 Na 1.2 Mg 0.4 Ti 0.1 P 0.1 Mn 0.1 H 0.1	1.8
Wood	Average		C 49.6 O 43.2 H 6.2 N 0.9	
	Ash			0.75
	Balsa			0.12
	Beech			0.80
	Birch			0.64
	Cedar			0.53
	Cherry			0.80
	Elm			0.57
	Hickory			0.76
	Locust			0.69
	Maple			0.68

	Oak			0.75
	Pine			0.55
	Poplar			0.42
	Spruce			0.59
	Walnut			0.67
Metal	Aluminum	EC-O	Al 99.45+	2.7
	Brass, red		Cu 85 Zn 15	8.75
	Bronze, commercial		Cu 90 Sn 10	8.80
	Copper		Cu 99.94	8.96
	Iron, cast gray		Fe 94 C 3 S 2 Mn 0.65	7.60
	Nickel		Ni 99.4	8.89
	Stainless Steel	S30400	Fe 68 Cr 19 Ni 10 Mn 2 Si 0.75 C 0.08 P 0.04 S 0.03	8.0
	Carbon Steel	G10400	Fe 99 Mn 0.75 C 0.4	8.0
	Tin-lead Solder	(Sn/Pb ratio varies)	Sn 50 Pb 50	8.85
	Zinc		Zn 99.9+ Pb 0.1-	7.1
Portland Cement	Type I		Ca 46.8 O 36.0 Si 10.1 Al 3.3 Fe 1.8 Mg 1.4 S 0.7	3.1
Concrete	½ Limestone and ½ Granite Aggregate	Type I cement + sand + ½ granite and ½ limestone gravel + water	O 52.0 Si 24.8 Ca 14.0 C 2.4 Al 2.4 K 1.2 Fe 1.1 H 1.1 Na 0.6 Mg 0.4 S 0.1	2.4

	Limestone Aggregate	Type I cement + sand + limestone gravel + water	O 51.8 Si 17.7 Ca 21.6 C 4.9 Al 0.9 K 0.2 Fe 1.1 H 1.1 Mg 0.5 S 0.1	2.4
	Granite Aggregate	Type I cement + sand + granite gravel + water	O 52.3 Si 31.8 Ca 6.3 Al 3.8 K 2.1 Fe 1.1 H 1.1 Na 1.1 Mg 0.3 S 0.1 Ti 0.1	2.4
	½ Limestone and ½ Granite Aggregate, reinforced	Type I cement + sand + ½ granite and ½ limestone gravel + water + Steel reinforcement	O 49.0 Si 23.4 Ca 13.2 Fe 6.7 C 2.3 Al 2.3 K 1.1 H 1.0 Na 0.6 Mg 0.4 S 0.1	2.4
Grout		1 part Portland: 3 parts Sand	O 56.3 Si 23.0 Ca 15.6 H 2.8 Al 1.1 Fe 0.6 Mg 0.5 S 0.2	2.2
Mortar	Type N	1 part Portland: 1 part Hydrated lime: 5 parts Sand	O 53.9 Si 25.2 Ca 14.2 Mn 2.5 H 2.2 Al 0.8 Fe 0.4 Mg 0.3 S 0.2	2.3
Hydrated lime		U.S.A.	O 40.3 Ca 32.5 Mn 24.7 H 2.5	1.4

Stone	Granite		O 48.6 Si 33.7 Al 7.3 K 4.5 Na 2.3 Fe 1.9 Ca 1.0 Mg 0.3 Ti 0.2 P 0.1 H 0.1	2.7
	Limestone	Average	O 47.5 Ca 34.0 C 10.5 Si 3.2 Fe 1.8 Al 1.1 Mg 0.6 K 0.5 Mn 0.1 Na 0.1 H 0.1 P 0.1	2.2
	Marble		O 47.3 Ca 39.3 C 11.7 Mg 0.3 Al 0.1	2.7
	Slate, Shales, and Clays	Average	O 49.7 Si 28.0 Al 8.8 Fe 4.6 K 2.9 Mg 1.6 Ca 1.6 Na 1.3 C 0.6 Ti 0.5 H 0.5	2.7-2.8
Water	4 C	Fresh Water	O 88.8 H 11.2	1.0
	Ice		O 88.8 H 11.2	0.92
	Snow		O 88.8 H 11.2	0.13
Plastics	Polyethylene	$[C_2H_4]_n$	C 85.6 H 14.4	0.9
	Polyvinyl Chloride	PVC: $[C_2H_3Cl]_n$	Cl 56.7 C 38.4 H 4.8	1.4

	Polypropylene	$[C_3H_6]_n$	C 85.6 H 14.4	0.9
Insulation	Glass fiber		O 46.4 Si 33.6 Na 10.4 Ca 5.7 Mg 2.4 Al 0.5	0.02
	Polyurethane foam		C 55.8 Cl 20.4 O 13.4 H 8.4 N 2.0	0.04
	Expanded Perlite		O 53.8 Si 35.0 Al 10.6 H 0.6	0.16
Other	Gypsum, Stucco		O 55.8 Ca 23.3 S 18.6 H 2.3	2.3
	Glass	Window	O 46.4 Si 33.6 Na 10.4 Ca 5.7 Mg 2.4 Al 0.5	2.5
	Paper		O 49 C 44 H 6	0.92

APPENDIX D: COMPONENT MATERIALS

Special Type	Breakdown	Description	Makeup	Chemical Makeup	Density (gm/cm^3)	Thickness (inches)
Interior Wall	Residential		2 x ½" Gypsum board 2 x 4 Pine studs every 16" 1¾" Steel nails 8" apart	O 54.7 Ca 21.5 S 17.2 C 3.9 H 2.5	0.56	4.5
	Residential Pipe	Above + 2 Copper and 1 PVC Pipes + 2 Copper Electrical Wires	2 x ½" Gypsum board 2 x 4 Pine studs every 16" 1¾" Steel nails 8" apart 2 x 2" Cu pipes (Type L) 1 x 4" PVC pipe 2 x 12 gauge Cu wires	O 52.4 Ca 20.5 S 16.5 C 4.2 Cu 3.2 H 2.4 Cl 0.8	0.58	4.5
	Concrete Block	Concrete block with steel ties	$7\frac{5}{8}$ " Lightweight concrete blocks (½ granite and ½ limestone aggregate) 6", 9 gauge Stainless steel ties every 16"	O 52.0 Si 24.8 Ca 14.0 C 2.4 Al 2.4 K 1.2 Fe 1.1 H 1.1 Na 0.6 Mg 0.4 S 0.1	0.76	7.6

	Concrete Block and Gypsum	Above + 2 Gypsum wall boards	7 5/8 " Lightweight concrete blocks (½ granite and ½ limestone aggregate) 6", 9 gauge Stainless steel ties every 16" 2 x ½" Gypsum wallboards	O 53.1 Si 17.7 Ca 16.6 S 5.4 C 1.7 Al 1.7 H 1.4 K 0.9 Fe 0.8 Na 0.4 Mg 0.3	0.94	8.6
	Reinforced Concrete	Concrete with steel reinforcement	7 5/8 " Reinforced concrete (½ granite and ½ limestone aggregate with steel reinforcement)	O 49.0 Si 23.4 Ca 13.2 Fe 6.7 C 2.3 Al 2.3 K 1.1 H 1.0 Na 0.6 Mg 0.4 S 0.1	2.4	7 5/8
Exterior Wall	Residential , Wood	Interior residential wall + exterior pine siding	2 x ½" Gypsum board 2 x 4 Pine studs every 16" 1¾" Steel nails 8" apart ½" Pine siding	O 53.5 Ca 19.4 S 15.5 C 8.5 H 2.8 N 0.1	0.55	5
	Residential , Wood , Insulation	Interior residential wall + exterior pine siding + polyurethane foam insulation	2 x ½" Gypsum board 2 x 4 Pine studs every 16" 1¾" Steel nails 8" apart ½" Pine siding 2" Polyurethane foam	O 52.4 Ca 18.9 S 15.1 C 9.8 H 3.0 Cl 0.6 N 0.2	0.57	5

	, Brick	Exterior brick layer	$3\frac{5}{8}$ " Brick and Type N Mortar 2 x ½" Gypsum board 2 x 4 Pine studs every 16" 1¾" Steel nails 8" apart ½" Plywood siding Steel ties	O 50.3 Si 21.2 Ca 8.0 Al 6.1 S 4.6 C 2.4 Fe 2.3 H 1.1 K 0.7 Na 0.4 Mg 0.3 Mn 0.3	1.1	9.1
	, Stucco	Exterior stucco layer	1¾" Stucco 2 x ½" Gypsum board 2 x 4 Pine studs every 16" 1¾" Steel nails 8" apart	O 55.4 Ca 22.5 S 18.0 H 2.4 C 1.7	0.96	5.9
	Concrete Block , Brick	Lightweight concrete blocks + Brick	3 ¾" Brick, type N mortar $7\frac{5}{8}$ " Lightweight concrete blocks (½ granite and ½ limestone aggregate) 9", 9 gauge, Stainless steel ties every 16" 5" x 5" x ¼" Steel angle	O 50.2 Si 27.3 Ca 8.2 Al 5.8 Fe 2.3 K 1.5 C 1.1 Mg 0.9 H 0.7 Na 0.6 S 0.2 Mn 0.2	0.92	14

	, Granite		3" Granite veneer $7\frac{5}{8}$ " Lightweight concrete blocks (½ granite and ½ limestone aggregate) 9", 9 gauge, Stainless steel twisted dovetail anchors every 2 ft^2	O 50.8 Si 29.0 Ca 7.7 Al 4.7 K 2.7 Fe 1.5 Na 1.4 C 0.9 H 0.8 Mg 0.4 S 0.1 Ti 0.1 P 0.1	1.4	11.6
	, Stucco		$1\frac{3}{8}$ " Stucco $7\frac{5}{8}$ " Lightweight concrete blocks (½ granite and ½ limestone aggregate) 9", 9 gauge, Stainless steel twisted dovetail anchors every 2 ft^2 2" Polyurethane foam insulation	O 53.4 Ca 17.8 Si 14.1 S 8.0 C 1.8 H 1.6 Al 1.4 K 0.7 Fe 0.6 Na 0.3 Mg 0.2 Cl 0.2	0.88	11.5
Floor	Concrete		4" Thick Concrete (½ granite and ½ limestone aggregate)	O 52.0 Si 24.8 Ca 14.0 C 2.4 Al 2.4 K 1.2 Fe 1.1 H 1.1 Na 0.6 Mg 0.4 S 0.1	2.4	4

	Concrete and Wood		4" Thick Concrete (½ granite and ½ limestone aggregate) ¾" Plywood ¾" Oak	O 51.2 Si 22.7 Ca 12.8 C 6.5 Al 2.2 H 1.5 K 1.1 Fe 1.0 Na 0.5 Mg 0.4 S 0.1 N 0.1	1.9	5.5
	Wood Floor over Gypsum ceiling		½" Gypsum board 2x8 Wood joists every 16" ½" Oak strips ¾" Plywood	O 49.4 C 25.1 Ca 11.5 S 9.2 H 4.3 N 0.5	0.27	8¾
Window	Glass, Single pane , Wood frame	Double hung	2 x (¼"x2'x3') Window glass 2 x (2x4)'s Pine Wood sill: 1.5" x 4.5" x 3' (Pine) 2 x frames: 47.3 in^3	O 45.7 Si 26.6 C 10.3 Na 8.2 Ca 4.5 Mg 1.9 H 1.3 Al 0.4 N 0.2	0.14	4.5
	, Double pane , Wood frame	Double hung with double pane glass	4 x (¼"x2'x3') Window glass 2 x (2x4)'s (Pine) Wood sill: 1.5" x 4.5" x 3' (Pine) 2 x frames: 47.3 in^3 (Pine)	O 46.0 Si 29.7 Na 9.2 C 5.7 Ca 5.0 Mg 2.1 H 0.7 Al 0.4 N 0.1	0.26	4.5

	, Single pane , Al frame	Sliding glass window	2 x (¼"x2'x2.5') Window glass Al sill: 4.4" x 5' x $\frac{3}{32}$ " 2 x Al frames: 51.8 in^3 each Al tracks: 2 x (1" x 5' x $\frac{3}{32}$ ") Al border: 25.8 in^3	Al 31.8 O 31.6 Si 22.9 Na 7.1 Ca 3.9 Mg 1.6	0.21	4.4
	, Double pane , Al frame	Sliding glass window with double pane glass	4 x (¼"x2'x2.5') Window glass Al sill: 5.9" x 5' x $\frac{3}{32}$ " 2 x Al frames: 51.8 in^3 each Al tracks: 2 x (2.5" x $\frac{3}{32}$ " x 5') Al border: 25.8 in^3	O 36.1 Si 26.1 Al 22.6 Na 8.1 Ca 4.4 Mg 1.9	0.27	5.9
Door	Wood , Solid	Solid oak door	6'10" x 2'8" x 1¾" Oak 3 Brass hinges and screws Brass and Steel door knob and lock	C 48.5 O 42.2 H 6.1 Cu 1.7 N 0.9 Zn 0.3 Fe 0.2	0.77	1¾
	, Hollow	Oak veneer door	1/16" oak veneer 3 Brass hinges and screws Brass and Steel door knob and lock Wood honey-comb core	C 36.5 O 31.8 Cu 20.8 H 4.6 Zn 3.7 Fe 2.0 N 0.7	0.14	1¾
	Steel		0.0598" Steel exterior 3 Steel hinges and screws Steel door knob and lock Polyurethane core	Fe 93.6 C 3.4 Cl 1.1 O 0.7 Mn 0.7 H 0.5 N 0.1	0.66	1¾

APPENDIX E: RADIONUCLIDE LIBRARY

QUEST includes an electronic version of the Erdtmann-Soyka gamma-ray library (1979). This library was compiled from the original source, with all of the radioactive nuclides and gamma rays based on the old "Blue Book" compilation. We have added the stable nuclides, neutron cross sections, and parent-daughter branching ratios for the natural decay chains of ^{238}U, ^{235}U, ^{232}Th, and some of the fission products. The relational database, contained in "`Quest\Database\quest.mdb`" is in Microsoft Access format, and consists of two tables: "Nuclides" and "Gammas". These tables are described in more detail below. Fields currently unused in QUEST are highlighted gray.

iZA	Energy	Br	BrError	BrCode	Brmap	Ag1	Ag2	Ag3	Ag4
100017	511	0	0		0	0	0	0	0
100018	511	2	0	A	1	1042.6	658.1	1700.7	0
100018	658.1	0.0012	0	A	1	1042.6	0	0	0
100018	1042.6	0.074	0	A	1	0	0	0	0
100018	1700.7	0.0005	0	A	1	1042.6	658.1	0	0
100019	511	2	0	A	1	0	0	0	0

iZA	$(Z \times 10^4)$ + isotope (A) + metastable flags [m (300), n (600), a (900), b (1200), c (1500)].
Energy	Photon energy (keV).
Br	Branching ratio, absolute / relative intensity (%).
BrError	Error in the branching ration (currently unused).
BrCodes	Descriptive codes given in Erdtmann/Soyka (Section 9, p. X-XI): absolute intensity, relative intensity, X-ray, unresolved doublet, complex line, weak line, uncertain transition, less-than.
Brmap	Binary representation of BrCodes: A = 1, R = 2, X = 4, D = 8, C = 16, W = 32.
Ag1-Ag4	Energies of the associated gammas (currently unused).

Table E.1: A portion of the Gammas table, together with field explanations.

Isotope	iZA	Hlife	iP1	iP2	iD1	BR1	iD2	BR2	iD3	BR3	ABUND	Th_Met	Th_Gro	Res_Me
Ne- 17	100017	0.109	0	0	90017	1	0	0	0	0	0	0		
Ne- 18	100018	1.67	0	0	0	0	0	0	0	0	0	0		
Ne- 19	100019	17.4	0	0	0	0	0	0	0	0	0	0		
Ne- 20	100020										0.9048		0.037	
Ne- 21	100021										0.0027		0.7	
Ne- 22	100022										0.0925		0.048	
Ne- 23	100023	38	0	0	0	0	0	0	0	0	0	0		
Ne- 24	100024	202.8	0	0	11002	1	0	0	0	0	0	0		

Res_Groun	Refs	Gen	GenMap
	74 SE 1		0
	72 AJ 1	CHA O 16,CHA NE 20	32
	68 LE 1	CHA F 19,NFA NE 20	34
0.02	STABLE	NAT	8
0.3	STABLE	NAT	8
0.02	STABLE	NAT	8
	68 LE 1,70 FI 1	NTH NE 22,NFA NA 23,NFA MG 26	3
	69 MC 2	CHA NE 22	32

Isotope	Isotope name: atomic symbol and Z.
iZA	$(Z \times 10^4)$ + isotope (A) + metastable flags [m (300), n (600), a (900), b (1200), c (1500)].
Hlife	Half-life (seconds).

iP1	iZA of first parent (currently unused).
iP2	iZA of second parent (currently unused).
iD1	iZA of first daughter.
BR1	iD1 first daughter branching ratio.
iD2	iZA of second daughter.
BR2	iD2 second daughter branching ratio.
iD3	iZA of third daughter.
BR3	iD3 third daughter branching ratio.
ABUND	Natural abundance.
Th Meta	Thermal neutron cross section to the metastable state (barns, currently unused).
Th Ground	Thermal neutron cross section to the ground state (barns, currently unused).
Res Meta	Resonance integral to the metastable state (barns, currently unused).
Res Ground	Resonance integral to the ground state (bars, currently unused).
Refs	References to data source given in Erdtmann/Soyka (Section 8, p.X).
Gen	Nuclide generators: NTHermal, NFAst, NFIssion, NATural, PHOton, CHArged particle..
GenMap	Binary representation of Gen: NTH = 1, NFA = 2, NFI = 8, PHO = 16, CHA = 32.

Table E.2: A portion of the Nuclides table, together with field explanations.

DISTRIBUTION:

Richard G. Hansen (3)
Bechtel Nevada, Remote Sensing Laboratory
P.O. Box 98521, MS RSL-21
Las Vegas, Nevada 89193-8521

Walter K. Hensley
Pacific Northwest National Laboratory
P.O. Box 999, MS P8-08
1 Battelle Blvd.
Richland, Washington 99352

1	MS 9402	T. O. Hunter		8000
		Attn:	J. B. Wright	2200
			J. F. Ney (A)	5200
			M. E. John	8100
			L. A. West	8200
			W. J. McLean	8300
			R. C. Wayne	8400
			P. N. Smith	8500
			P. E. Brewer	8600
			T. M. Dyer	8700
			L. A. Hiles	8800
			D. L. Crawford	8900
1	MS 0451	R. E. Trellue		6238
1	MS 0571	D. J. Allen		5914
1	MS 0571	D. R. Waymire		5914
1	MS 0744	R. G. Cox		6412
1	MS 0759	M. K. Snell		5845
1	MS 0767	R. K. Wilson		5135
1	MS 0769	D. S. Miyoshi		5800
1	MS 0970	J. R. Kelsey		5700
1	MS 1131	M. B. Sandoval		5849
1	MS 1165	J. Polito		9300
1	MS 1207	R. W. Moya		5908
1	MS 9103	G. A. Thomas		8120
1	MS 9201	L. D. Brandt		8112
1	MS 9201	P. K. Falcone		8114
1	MS 9201	M. E. Goldsby		8114
40	MS 9201	M. M. Johnson		8114
1	MS 9201	T. H. West		8114
1	MS 9214	L. M. Napolitano		8130
1	MS 9214	T. D. Plantenga		8950
1	MS 9405	J. M. Hruby		8230
1	MS 9405	R. B. James		8230
1	MS 9409	T. L. Porter		8260
1	MS 9420	W. B. Wilcox		8220
3	MS 0161	Patent and Licensing Office, Organization 11500		
4	MS 0899	Technical Library, 4916		
3	MS 9018	Central Technical Files, 8940-2		
1	MS 9021	Technical Communications Department, 8815/Technical Library, MS 0899, 4916		
2	MS 9021	Technical Communications Department, 8815, for DOE/OSTI		