A Simple Interface for Building Environment Simulation Codes

by

Charles R. Broderick, III

B.S., Computer Science and Engineering (1999)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING

in

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 19, 2000

Author_____
Department of Electrical Engineering and Computer Science
May 19, 2000

Certified
by_____
Qingyan Chen
Thesis Supervisor

Accepted
by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Simple Interface to Building Environment Simulation Codes

by

Charles R. Broderick, III

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 19, 2000

Abstract

Most airflow simulation codes have their own user interfaces for constructing models and visualizing results. These interfaces are often too complex and too confusing for the architect who has no experience in CFD simulation. The Simplified CFD Interface (SCI) system is an improvement on other airflow simulation interfaces. It is intuitive enough for CFD novices to quickly build and visualize airflow models, and powerful enough to interface with multiple CFD solver programs. The SCI design and implementation are discussed.

A·C·K·N·O·W·L·E·D·G·E·M·E·N·T·S

The author would like to thank Yan Chen, Jelena Srebric, and John Zhai for their technical contributions and support.

ΤΚΦ to Owen Williams, Ben Krinsky, Mike Terry, and Dennis Gregorovic.

The author would also like to thank Marielle Yohe-Williams, Emma Teng, Col. and Mrs. Charles R. Broderick, Jr., and Mr. and Mrs. James D. Reynolds for their support.

CONTENTS

1. Introduction to Computational Fluid Dynamics

Fluid motion is an important physical phenomenon for many engineering applications. These include airflow around and inside buildings, the aerodynamic response of aircraft, and weather patterns. The Navier-Stokes equations model momentum conservation for fluid flow, and therefore the solution of these equations provides important insight into fluid design issues. Computational Fluid Dynamics (CFD) is a popular numerical method for the solution of conservation laws such as mass, momentum and energy conservation, and can be used to approximate solutions to Navier-Stokes.

1.1 Building a CFD Compatible Model

Fluid flow problems require a three step transformation into a special framework before CFD methods can be used to solve them. The first step is to choose a finite set of connected "nodes" on which the Navier-Stokes equations will be satisfied. Each node is defined by its physical location within the problem domain. Once the solution is computed, the flow values at each of these locations is known to within a certain precision. This collection of nodes is also known as a computational mesh.

Choosing the most useful geometry for a computational mesh is the subject of much academic research. In general, the more dense the mesh is, the more detailed results are returned. However, more nodes in the mesh results in more complexity, which in turn extends the time required to achieve an acceptable level of solution accuracy. Custom meshes with low nodal density in areas of no interest and high nodal density around interesting phenomenon are very common compromises.

The second step in creating a CFD model is to define the location and properties of the flowing materials. These include parameters such as density, heat conductivity, temperature, and plasticity. Since the smallest geometric volume is defined by the space between nodes in the mesh (also called "control volumes",

"zones", or hexahedrons), materials are placed in the mesh by listing which control volumes they inhabit[1]. It should be apparent that the smaller the control volumes are, the more accurate the representation of the actual geometry becomes. However, smaller control volumes requires a more dense mesh, which in turn leads to slower computation times.

The final step in generating a CFD-compatible model is to specify any outside forces that influence the fluid flow. These forces are integrated into the CFD model through a list of boundary conditions. For example, the wind generated by the fan at the front of a wind tunnel would be defined by a single boundary condition. The effect of boundary conditions are factored into the Navier-Stokes equations by the numerical solver.

1.3 Airflow Simulation

Mastery of CFD simulation was once relegated to the disciplines of mechanical engineering, material science, hydrodynamic physics and aerodynamics engineering. Recently, the field of architecture has begun to recognize the design benefits that they could reap by converting their blueprints into CFD airflow models. Metrics such as indoor air quality, thermal comfort, and building ventilation effectiveness can be calculated and analyzed. This shift was most likely brought about by the eroding cost of powerful computation, the successful market penetration of CAD tools, and the increasing sophistication provided by numerical simulation.

One of the strongest players in the CFD space is a company called CHAM. Founded by Professor Brian Spalding, a former Professor of Heat Transfer at the London Imperial College of Science and Technology, CHAM boasts ownership of the "most widely-used CFD code in the world." Called PHOENICS (Parabolic

---

[1] Some CFD programs support the notion of a "mixed zone" which can contain two or more materials in a single zone. While the boundaries between the materials in a single zone are not calculated, the percentage of each material inside the zone is maintained.

Hyperbolic or Elliptic Numerical Integration Code Series, the first three words referring to its different methods of numerical approximation), it was originally written in FORTRAN in 1981. Since then, it has gone through multiple expansive rewrites and is now available in PC and workstation version.

Today, PHOENICS is packaged as a full-service suite of CFD solvers and tools. CFD model creation, as specified in chapter 1.2, is handled by the helper application PHOENICS-VR. PHOENICS-VR allows a user to interactively create a computational mesh, define materials, place objects into the model, and define boundary conditions that impact the model. The resulting model definition is saved to disk in the form of a text file, which is subsequently processed by the CFD solver EARTH. EARTH saves its solution in a large binary file, which can be visualized on-screen by an application called PHOTON.

There are four flavors of plots that PHOTON can create: vector plots, contour plots, iso-surfaces, and streamlines. Vector plots identify the air speed at each node with a directional arrow. Contour plots trace lines of constant value along planes of scalar data. Iso-surfaces trace surfaces of constant value throughout a volume. Streamlines follow vector data around a volume, starting at a chosen point, to indicate the likely path of a released particle.

The language of PHOENICS is best suited for the advanced CFD designer, and although there is evidence that CHAM has attempted to simplify the simulation process for the "CFD-illiterate", the generality of the program suggests that these improvements will be met with only minor success. Even experts familiar with CFD mathematical foundations might lack the obtuse FORTRAN knowledge that permeates throughout the application.

2. Motivation

2.1 Defining the Typical User

A general practice architect does not typically have the time, inclination, or mathematical foundation to learn how computational fluid dynamic methods work. However, she is familiar with using personal computers, and perhaps has some experience with computer drafting tools. The more technically advanced architect is more comfortable with building environment concepts. She knows how to turn air comfort analysis results into design alterations, and is familiar with model generation in architecture software. Both the general practice architect and the technically advanced architect have access to a Microsoft Windows personal computers.

A "Building Technologist" is an architect who is highly skilled in technological and engineering issues. She most likely holds a degree in mechanical engineering in addition to architecture. She understands both the advantages and the disadvantages of CFD analysis pertaining to building design. She has done CFD analysis before, and knows how to coerce a CFD model to converge with accurate results.

Programs such as CHAM's PHOENICS are best used by this category of architect, for they are suited to understand and appreciate its plethora of features. The general practice architect, on the other hand, would be baffled and intimidated by the tasks that PHOENICS requires. This explains why CHAM employs a cadre of billable consultants who specialize in using CHAM software to help confused clients perform computational analysis.

This leaves wide open the opportunity to create CFD software geared towards the general architect.

2.2 Problems With Current Systems

PHOENICS and similar systems suffer from the same flaw: they were originally written for CFD researchers and building technologists. Efforts to make using these programs easier for novices to use have met with limited success[2]. The source of the problem is most often a complicated, cluttered, and poorly designed user interface. The lack of a simple interface for the general architect is a substantial problem.

The fractured marketplace has provided another source of user frustration. While the CAD market has solidified and standardized, the younger CFD market still suffers from industry-wide compatibility issues. For example, the switching costs involved in using a new CFD package is high because the user must re-enter old models with the new interface.

High switching costs are also incurred in the time spent learning to use a different user interface for each airflow simulation code. Each CFD system tends to have its own home-grown methods for user interaction, and some programs, like PHOENICS, have three or four. The lack of a common CFD interface flexible enough to handle multiple CFD solvers slows innovation and proliferation in solver design.

3. SCI High Level Design

We endeavored to create an interface system to fill the void left by more sophisticated CFD tools. We call our interface SCI (pronounced "sky"), for Simplified CFD Interface.

---

[2] See MICA Final Review Meeting Results, by Professor Brian Spaulding, CHAM Founder and CEO, http://www.cham.co.uk/website/new/mica/micafi.htm

3.1 Design Principles

SCI is a user interface front end for both indoor and outdoor airflow simulation software. It was designed from conception to be immediately useful anyone in the non-technical architecture community who wants to make CFD analysis part of their design process. This goal lead us to the following set of design principles:

Make It General. The diversity of existing CFD solvers is significant enough to warrant creating an interface that can work with more than just one or two. The variety in CFD solver systems stems from innovations in both CFD technology and in evolving environment analysis metrics. If users become locked in to SCI's interface design, there will be less trepidation about trying new solvers.

Keep The Interface Simple. When building an interface to a CFD solver, the easiest mistake to make is to integrate the hundreds of computation-modifying solver parameters into the editor. Our users are not familiar with these parameters, and we would like to minimize the amount of learning required to use SCI. Most features fail the 70/30 metric of usefulness (70% of all indoor/outdoor airflow problems can be solved by using 30% of a solver's parameters) and have minimal impact on the bottom line analysis.

Aggressively Integrate Format Standards. Forcing a user to re-enter a model that has been already been constructed for another program is unproductive. Fortunately, the CAD market has gone through similar file format growing pains and the CFD sector can benefit from their success.

Follow the Architect's Process. We would like to focus on guiding the user through the CFD process in a natural, intuitive way. This often requires making intelligent assumptions about the input model on behalf of the user.

Simple Hardware Requirements. Complicated CFD models containing upwards of five millions nodes require massive computational resources that small-to-medium architecture firms can not afford to buy or maintain. We will concentrate on technologies and solvers within the reach of moderate computational power.

One-Stop Shopping. PHOENICS has an entire suite of programs and analysis tools in their general package distribution. We would like to remove most of these extra, separate utilities and instead build an all-in-one application. The final deliverable should be able to generate, run, and analyze models independent of other software.

Make It Secure. Our system is intended for broad educational and professional use. To keep its distribution under control, we would like to engineer a security system that requires contact with a central location before the system is activated.

These principles are used throughout the design and implementation of SCI, and are the standard by which we evaluate our system's success. New features and extensions were created in response to these principles, and other features were thrown out for violating them.

3.2 Developing a System Diagram

Architects use CFD analysis to help them understand the effects different building designs have on complex environment metrics. These results are used to motivate intelligent modifications to designs. This suggests a process that a CFD user is likely to follow:

1. Enter a design into the computer
2. Enter the necessary information that will allow the CFD program to converge (specify the computational mesh, boundary conditions, etc.)
3. Run the CFD program.
4. Did the program converge? Yes: Go to step 5. No: Go back to step 2.
5. Visualize the results.
6. Think about the quality of the results. Is it good enough to make rational design decisions with? Yes: Go to step 7. No: Go back and do step 2.

7. Think about the implications of the results. Do they indicate a satisfactory design? Yes: Save the results as proof. No: Use the results to make a intelligent change in the design and start over.

To begin, we will identify all the subsystems that fall out of this process.

We will need to build a subsystem that allows the user to enter 70% of all indoor and outdoor airflow designs. This indicates we will need a user-interface subsystem that gets the relevant information from the user. We will need it to accept boundary conditions, meshes, and a minimal set of computational flags and variables.

In order to successfully integrate with different CFD solvers, we will need a subsystem that translates the internal SCI data model to the various solver input format. This system will also need import results into the internal data structures so the visualization subsystem can use it. We will call these systems "solver translation services".

When the computation is completed, the architect will want to view her results. This suggests that we will need a visualization subsystem. This includes an arbitrary viewing module that interacts with the user, a three-dimensional graphing subsystem, and a user-interface module that allows the user to select what kinds of visualizations to perform. Most users will want a basic selection of plot methods to visualize their data. These include geometry plots, scalar contour line plots, scalar hot-to-cold color fields (also known as pseudocolor plots), vector plots with directional arrowheads, and computational mesh plots.

We will need a subsystem that can import and export geometry file formats from other applications. This subsystem should handle the translation between SCI-specific data structures and foreign data structures. If an important file format is proprietary, then some portions of this subsystem might need to be purchased from other corporations or entities.

We will need to build a subsystem to handle persistent data storage to access permanent storage devices. This system will create durable records of models and results, which can later be recalled for further analysis and review.

We will need a two-pronged security subsystem. One half of the system will be attached to the SCI deliverable that is distributed to the clients. This part will require evidence that the user has checked in with the central location before permitting access to SCI. The other part will be kept at the central location, to create and distribute "keys" to users who register.

We can organize all these subsystems by identifying common themes that connect them. There appear to be three: visualization oriented subsystems, model generation/user-interface subsystems, and program/file infrastructure subsystems. In addition to these, SCI needs the support of a few external systems to operate. These include the operating system, CFD solvers, and (possibly) proprietary file format support systems.

3.3 Examining the Design

With the system design in mind, lets return to our design principles and take a closer look at them.

- **Make It General**

Creating an interface that is useful to more than one CFD solver requires thinking about which model parameters are basic and necessary. We have already identified a few common parameters: computational meshes, model environment qualities, and computational environment qualities. The user-interface subsystems will have to define which individual parameters will be included and which ones are too specific or too esoteric to bother the user with.

The subsystem that translates SCI's internal data into an input format readable by a CFD solver is crucial to SCI's success. It will be the universal gear that makes our interface general to more than one solver. The design of this subsystem is critical to the success of this design principle. Let's pretend the particular solver we want use with SCI requires more information than SCI permits the user to enter. This will be a common situation, as our interface tends to restrict rather than enrich computational specificity. How will we deal with such a situation?

We can think of a solver program as type system, containing a single function with an abnormally long signature of about fifty or so arguments. Our caller procedure (SCI) might only has enough data to provide for, say, half the number of requested arguments.

C++ deals with this situation by allowing the called procedure to specify parameter defaults in the function prototype. This would be a poor and impractical solution to implement in SCI. For one, access to the CFD solver's input code is impossible, and we cannot assume that the programmer has provided for the case when expected data is not provided. Also, SCI is in a much better position to make an intelligent guess to the value for a parameter than the solver, since it has access to additional model information the solver might not be aware of.

There is considerable intelligence present in the solver translation services subsystem. Writing a translation service for a new CFD solver usually requires contact with the CFD programmer to discuss the best default values to choose when the user is not given the opportunity to directly specify. Occasionally, this will require SCI to make a decision dynamically, but more often than not it will be a static decision that SCI will make in all circumstances.

It behooves us to create an infrastructure whereby the ease with which these translation subsystems are written is maximized. This will encourage CFD programmers to include a translation service when they write new solvers. This creates a situation where everybody wins. The CFD programmer gets an interface customers are familiar with, removing the burden of having to write her own. SCI users get to enjoy the benefits of a CFD market constantly creating new solvers for their interface.

SCI will often limit the abilities of a solver by limiting its input domain. Why would CFD programmers want to use an interface that does not take full advantage of its unique aspects?

The answer comes in two parts. First, a CFD solver's unique features might not require new information. For example, the innovation could be a faster way to solve fluid flow in buildings with a certain floor plan, or it could be a more precise equation for solving turbulent fluid flow. None of these innovations require additional model information from the user. Second, SCI's design principles imply that the architect probably does not have access to, or know how to come about, the additional information a CFD solver is requesting to take advantage of the new feature.

- **Keep The Interface Simple**

To keep the interface simple, we need to simplify all of SCI's subsystems that interact with the user. We have previously discussed a method of keeping these systems simple by removing as many computational details of CFD as possible. We can also make the interface simple by making it intuitive. This can be achieved in two steps: one, figure out what kinds of things the user is already adept at doing and do similar things, and two, figure out what the user expects and deliver on those expectations

The Windows operating system has a standard user interface toolbox that the vast majority of Windows applications employ. This toolbox is known as the Microsoft Foundation Class, or MFC. It provides the

graphical look and feel that has become the primary expectation for most personal computer users. Some examples of expected program behavior include the resizable window, the pull down menus from a textual, horizontal list on the top of a window, the three standard minimize/maximize/kill buttons on the top right of a window, the icon toolbar, and the operation of the "Open File" dialog box.

Windows users are also unconsciously familiar with the single/multiple document paradigm. This is the concept of an application "operating" on a single or set of unique "documents". The documents contain some form of individual data that can edited, created, viewed, or otherwise processed by that application. Since documents are kept as separate files in the file system, the user quickly draws a distinction between application and data. This system gave birth to the concepts of "New Document", "Save Document", and "Open Document" that we usually take for granted.

SCI is written and compiled on the Windows family of operating systems with the Microsoft C++ development environment (Visual C++ 6.0). This allows us to inherit both the Windows look and feel and the single/multiple document paradigm without exerting excessive programming effort.

SCI considers every aspect of each CFD model to be its own separate "document". This includes the model's geometry, its boundary conditions, its computational and model environment parameters, the current visualizations on the screen, and any results that may have been computed. Thus, selecting a "New" document will dump the current model (after asking if the user would like to save), and generate a fresh, blank model that is waiting for new geometric and computational parameterizations. Similarly, selecting "Save" will cause the persistent storage system to save all model information into its own unique document.

A simple interface delivers what the user expects. For example, 2-D navigation of images has become a common computer application task. Adobe Photoshop popularized the navigation hand and the magnifying

glass, where a control-left click and drag with the mouse results in a two-dimensional image translation, a right click results in a zoom-in, and a control-right click results in a zoom-out. We will use the same images and functions in our visualization system.

- **Aggressively Integrate Format Standards**

Airflow is not the only environment analysis tool that architects use. In fact, there are a series of building technology analysis available to the architect. Some tools aid in design while others focus on building efficiency.

Sometimes these tools are brought to bear on a single project. When this happens, similar data is required for input and it may be desirable to share this data among a group of applications. For example, here is a system design for the interoperability of an airflow simulator, a CAD application, a pipe design tool, and an energy analysis tool (EnergyPlus).

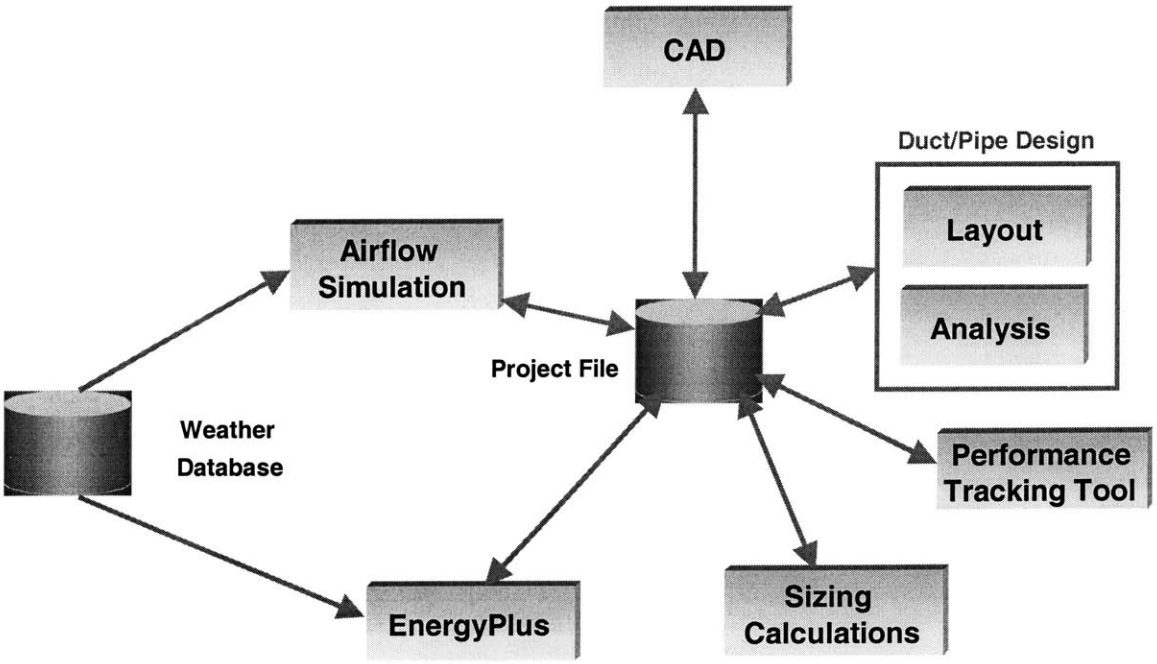


Figure 1: Data flow diagram for multiple-analysis system using same project file.

A common technique to allow multiple applications to communicate together is to use an agreed-upon persistent data system, such as a database or a universal file standard (XML, for example). From the figure, you can tell that the project file serves as the central storage and retrieval point for all the other applications. Thus, it is critical that this system be designed with the full participation of all the programmers. Designing this API is not a trivial task, and it requires many negotiations between designers with many different requirements. However, the rewards are high. If successful, the whole community can rally behind a system and begin enriching their tools through integration. The more tightly integrated the analysis market is, the more useful and marketable the product become.

The Industrial Foundation Class, or IFC, is a persistent data storage API built for the building environment simulation community. It has been created through a combined committee of both academia and industry, and already has an implementation. The API design work is superb, and the software community should benefit from it for years to come.

- **Follow the Architect's Process**

We would like our application's process to be as intuitive to our users as possible. To accomplish this, we need to understand what our users are trying to get out of our tool. We will brainstorm on this subject by working through a simple exercise. It is called the Flowchart Boundaries Framework, and answering its questions will help us focus on what the program flow should feel like:

1. What materials does the user start with?
2. What materials does the user end with?
3. What kinds of decisions should the user be making during the program execution?

Our users start with the geometry of a physical building or group of buildings, either in paper (blueprints, design sketches, etc.) or electronic (CAD or IFC files, etc.) form. There may or may not be environmental data such as average temperature at the build site or average wind flow around the problem boundaries. In case there isn't, SCI should be prepared to have reasonable default parameters set up.

The users of SCI eventually want to look at how air, contaminants, relative humidity, temperature, pressure, comfort metrics, or other environment metrics are distributed at different locations around their designs. Preferably the user will want to view this information graphically, but it is possible that the user might be interested in the numerical results as well.

The user will need to decide a number of things during the run:

1. What kind of CFD solver should execute the model.
2. What kind of computational mesh to use.
3. What to set various environmental and computational parameters.
4. Which results to look at and how to look at them.
5. Whether or not to modify the geometry or otherwise tweak the model

Later, we will look at how the three major subsystems ask these questions of the user.

- **Simple Hardware Requirements**

The greatest boon to CFD work in the past two decades has been Moore's Law. Moore's Law was a harmless observation in 1965 by Intel chief Gordon Moore that microchip capacity seemed to be doubling every 18 months. The same was also true of processing speed. Moore's Law has continued to hold true ever since. This has put the power of reasonably detailed CFD analysis onto the desktops of most architecture firms.

MIT's Building Technology Laboratory has developed some reasonably fast CFD solvers that can run medium sized problems in an acceptable timeframe on a medium priced personal computer. SCI will initially be using these solvers for its beta release.

CHAM and some other leading providers of CFD solutions have suggested using the Internet as a client/server vehicle. The user would interact with powerful equipment through a web or Internet-enabled application. This is an appealing idea. We would like to leave the link between SCI and the solver flexible enough to accommodate for this future possibility.

- **One-Stop Shopping**

The trend in building environment simulation has been towards linking several tools together to make mutually beneficial alterations in designs over several metrics. This trend is being accelerated by joint industry operations, such as the development of IFC. Examining the logical conclusion of this trend suggests that all simulation tools will run under a single blanket environment, like SCI endeavors to do for airflow simulation, and not under separate stand-alone applications who use a common project file.

This goal is at least ten to fifteen years in the future. In the meantime, there is a opening in the community for a solid, complete interface environment such as SCI that can unite the airflow market. We will want all

the supporting tools, such as visualization, model generation, persistent storage, and parameter setting, to be included at this unified level.

4. SCI Low Level Design and Implementation

This chapter will discuss at length the issues, algorithms, design decisions, and features that went into building SCI. SCI is the third major revision of a code started in 1997 by the author, and has reached a length of 26,000 lines of C++ code, contained in 34 source files, implementing over 70 classes. It runs on all current families of Microsoft Windows equipped personal computers (Windows 95/98/00 and Windows NT).

4.1 Object Orientated Design

An object is a computational abstraction, referring to a type instantiation in an object-orientated environment. Object-oriented programming, or OOP, gained momentum in the 70's and 80's as a natural way to capture code reusability (through class inheritance), data abstractions (through abstract types), and implicit, natural design structure. An object typically consists of data members and functions that act on them.

SCI uses objects as the backbone of its design. There are two main reasons for this choice. One, our system diagram fits neatly into an object design. Each of the three main subsystems (visualization, model generation, program infrastructure) can be turned directly into either a class or a cluster of related classes. In other words, objects are a good way to provides the "skin" that will keep our systems recognizable and distinct.

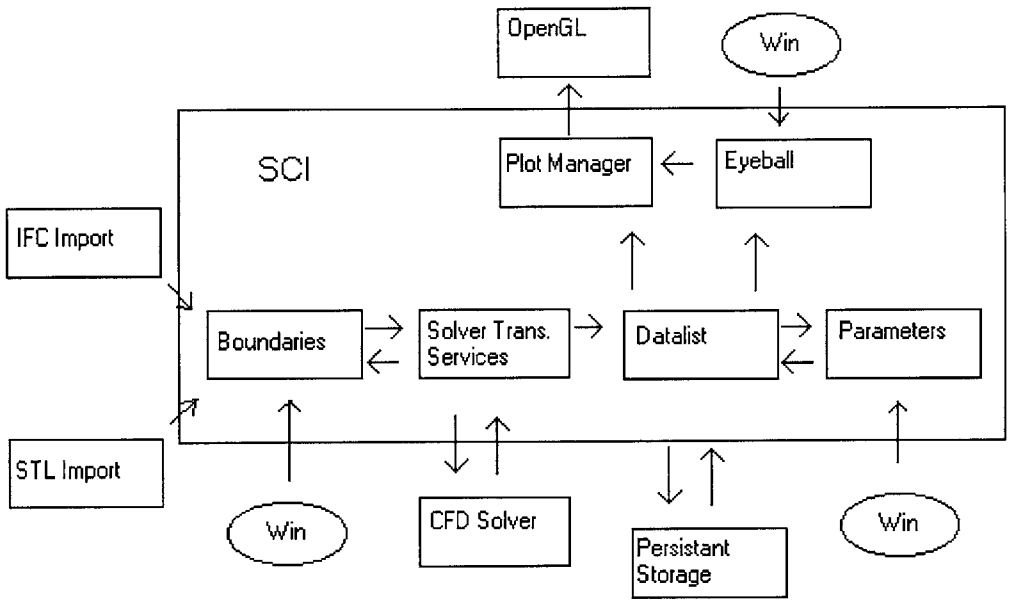
The second reason motivating an OOP design approach is MFC's use of object design. MFC objects allow application programmers to subclass critical window control code in a straightforward manner, making

windows programming much easier to implement. MFC objects are tightly integrated into Microsoft's C++ environment (Visual C++), which SCI is developed on. Given the rapidly evolving Windows environments, choosing to write Windows application without MFC support would be a costly mistake.

4.2 General Data Flow Diagram

The general data flow diagram (GDFD) presents a clear picture of how data flows through a system. It is a directed graph with classes represented by nodes and connecting edges indicating the possibility of data flowing between classes.

Figure 2: General Data Flow Diagram for SCI



The large rectangular box with SCI in the top right corner contains all the classes within the jurisdiction of our design. Interaction with outside objects and data sources is indicated by edges that pass through this membrane.

Following the arrows into the solver translation class indicates where the data necessary for the solver to work is gathered from. The datalist object controls all the model data – computational parameters (aggregated by the objects "problem parameters" and "iteration control"), computational mesh, and environmental model parameters (taken from the "property specification" object). The boundary manager controls the problem's geometric layout and boundary conditions. Together, these two data sources comprise the information necessary for a CFD solver to run.

Once the CFD solver returns with data, the only place for this data to flow is back into the solver translation class and into the datalist object. The datalist keeps track of all data returned by CFD solvers. Objects that need access to this information, such as the plot manager and the eyeball (for visualization purposes, see chapter 4.5), require data to flow from the datalist.

The plot manager object is in charge of placing visualizations on the screen at the user's request. Since the user may want to view the model layout as well as the solver results, the plot manager needs a data flow from the boundary manager as well as from the datalist object. When the user requests a visualization (accomplished through Windows), the plot manager aggregates the appropriate data, organizes it according to how the user wants to view it (using the eyeball object), and pipes this information to the OpenGL routines through the OpenGL API.

OpenGL is an open graphics standard distributed with Windows 95/98/00 and NT, and is supported by virtually every graphics card on the market. It has a well-written, open API and operates with the right abstraction of graphics primitives for our needs. We will be discussing how SCI renders CFD data into OpenGL graphics primitives when we discuss the implementation of the plots in chapter 4.5.

SCI uses the boundary manager to keep track of the model layout. This is why data needs to flow from geometry file importing classes like STL and IFC into the boundary manager. However, it is not a

requirement to import geometric data from these objects. A sufficient model geometry can be constructed through the Windows interface (indicated by the Windows input arrow) without importing files. This is an important point to remember about the GDFD – an edge indicates only that the design allows data to flow, not necessarily that data must flow.

There are four places where the Windows user interface can inject data into SCI. They are labeled on the GDFD as "Win." Windows gives data to these classes through MFC callbacks that these classes register to receive. When Windows determines that a mouse movement or a keyboard input is affecting one of the windows controlled by an object of these classes, an "event" is sent to the object containing both the information received and the context it was received in. This is why Windows applications is sometimes referred to as "event-driven programming".

The persistent storage system is shown to flow data to the entire SCI box. This is to preserve clutter in the diagram. In reality, persistent storage needs to access state information from each of the classes inside the SCI environment in order to execute a document save, and needs to send state information to execute a document load. This would needlessly complicate the GDFD by requiring two-way edges to connect each object inside the SCI box with persistent storage.

4.3 Model Generation Classes

The first step in running a CFD analysis is to specify the model. The models we want to generate should be specific enough to handle most of the airflow problems our users might want to solve. We will break down the task of model definition into three steps:

1. Define the geometry.
2. Define the computational mesh.

3. Specify the model parameters.

Furthermore, we would like our users to feel this process is simple and painless. To do this, we will need to implement intuitive systems. We defined our notion of user intuition in chapter 3, using the Flowchart Boundaries Framework and the design principles. That analysis indicated that we should capitalize on previous user experiences and focus on the materials available to the user.

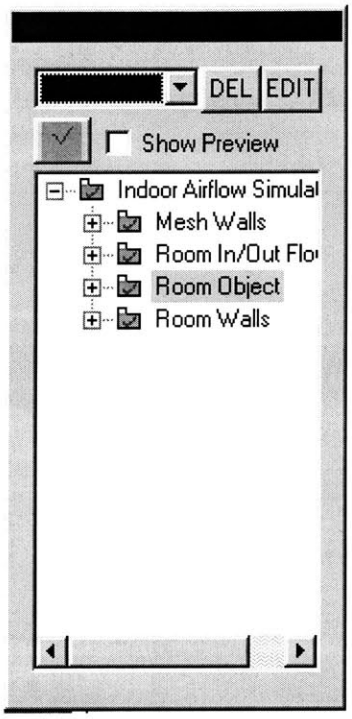
4.3.1 Defining the Geometry

There is an infinite number of geometric building blocks that we could use for model definition. For example, curved objects and polygonal structures are useful to some architects who focus on particular building structures, such as smokestacks. All of these objects, however, can be approximated by rectangular objects without any major penalty in precision. Therefore, SCI uses only rectangular-shaped objects of two or three dimensions to define the geometry of a room or building.

Objects like tables, chairs, and buildings that are placed in a computation mesh are represented as boundary conditions. Thus, "boundary manager" is the name of the class that controls all the actions related to geometric definition. The boundary manager object does not actually contain the boundary conditions, however. Rather, it controls access to the object that does: the boundary dialog object. This object stores all the boundary conditions entered by the user. It is subclassed from the MFC class CDialog, so in addition to managing the boundary conditions, it also handles Windows events.

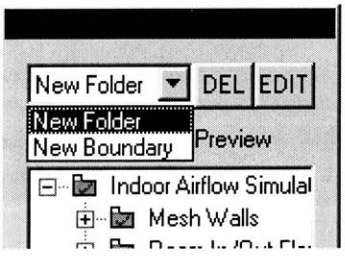
The boundary manager can launch the boundary dialog object (by registering it for Windows callbacks), move the position of the dialog object's window, and clear the dialog object of all its boundary conditions. This manager/dialog object relationship is also used by the plot manager/plot dialog objects. See chapter 4.5 for details.

The boundary manager dialog object visualizes the list of boundary conditions that have been defined so far. The box in the middle of the window is a list of these conditions. Boundaries are organized in a folder/file hierarchy, much like the file system of a hard drive. Folders are identified by a folder-shaped icon next to its name, and boundaries are identified by a box shaped object. Each boundary represents a single 2D or 3D rectangular shape.



Occasionally, the user will want to define objects, but not want them sent to the solver for inclusion in the model. This allows the user, for example, to switch between several different room configurations without having to maintain them in separate documents. A boundary object is included in the computation if the icon representing the boundary (or the folder) is colored green. A boundary object is not included in the computation if it is colored red. If a boundary is included below any folder that has been switched off, then that boundary condition will not be considered in the computation. The user activates and deactivates boundaries and folders by clicking a large green and red check button above the boundary view, next to the "show preview" button.

Figure 3: Boundary Manager



To delete a boundary folder or boundary object, users select the object and click the DEL button. To add a new boundary condition or folder, the user first selects the folder that will contain the item, and then clicks on the dropdown list box. There, the user can select either a new folder or a new boundary condition.

Figure 4: Add New Boundary

To change the type, location, or parameters of a particular boundary condition, the user first selects the boundary condition she wishes to view or edit, and then clicks on the EDIT button. This launches a second dialog box, which we will call the "Boundary Edit Dialog Object".

The boundary edit dialog object allows the user to specify the location, type, and parameters associated with a single boundary condition. Defining how the user can execute this task cements our notion of what a boundary condition is. This will have an immediate impact on the solver translation services – the broader the ability to specify a boundary condition, the more solvers the translation services can support.

We surveyed the different kinds of boundary conditions commonly used in airflow simulations and chose to provide for six different types: blockages, walls, inlets, outlets, symmetric planes, and user-defined sources.

The first two, blockages and walls, are the primary geometric building blocks for SCI models. A blockage is used to specify a solid, three dimensional rectangular shaped object that can have contaminant, airflow, temperature or heat flux associated with it. A wall is a two dimensional rectangular object that can have temperature or heat flux associated with it.

The other four boundary conditions (inlets, outlets, symmetric planes, and user-defined sources) are computational boundary conditions that modify how the CFD solver will compute results, and do not affect the geometric layout of the model.

We define inlets and outlets to be two dimensional rectangles. Inlets can have a variable rate of fluid flowing from them, a specific amount of mass, a temperature, and up to four different concentrations of

contaminants. Outlets are parameterized similarly, except a constant pressure is specified instead of a mass value.

Symmetric planes are a method of exploiting symmetries in a problem geometry. They are defined over a two-dimensional plane, and should ideally be positioned on the symmetric edge of a model (it makes little sense to put them anywhere else).

A user-defined source is a defined exactly like a wall, but it is not solid, and will not block the flow of fluids.

The boundary edit dialog object allows the user to enter all of this information. It is organized through a tab control, which lets the user select one of four possible attributes to edit – location, contaminants, airflow, and temperature/mass. Above the tab control is a dropdown which lets the user select the type of the boundary condition.

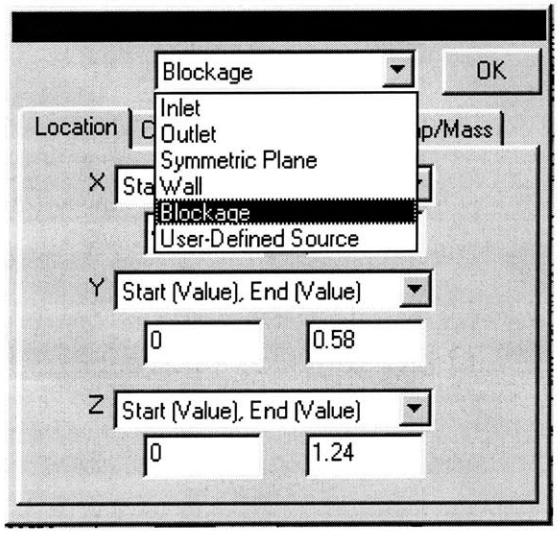


Figure 5: Boundary Edit Dialog Box

Changing the type of boundary condition alters the kinds of parameters you can specify. For example, a symmetric plane does not have any airflow associated with it. Therefore, if the user has selected "Symmetric Plane" as the type, there will be no edit boxes under the airflow tab.

To allow for flexibility, the user is given the choice of selecting eight different methods for specifying the location of each boundary condition. The default method (Start (Value), End (Value)) is to specify where

each side of the rectangle or box lay along each dimension. If a 1 m^3 square box resides at the origin, then the box would be entered by choosing to specify the "Start (Value), End (Value)" option for each dimension, and entering 0 and 1 in the six edit boxes.

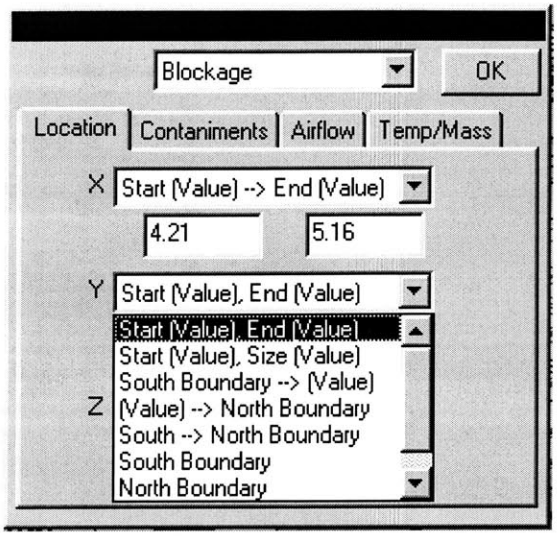


Figure 6: Boundary Location Options

All boundaries must be specified by defining their location along each dimension (X, Y, and Z) separately. However, the user does not have to give fixed locations for each boundary. Some boundary conditions (typically the non-geometric kinds, like the symmetric plane) work best if they are defined along the extremes of the mesh. The computational mesh may change shape at the users request, and it is an annoying to constantly go back to the boundary editor and update the locations of these conditions.

To solve this problem, the user is given the opportunity to specify a boundary that is dynamically attached to the extremes of the computational mesh. For example, suppose the user wanted to specify a plane of heat that sliced through the entire XY plane of the mesh, two meters above the floor. She would create a new boundary in the boundary manager dialog object, select it, and click on the EDIT button to bring up the boundary editor dialog object. Then, she would select "User-Defined Source", and specify a temperature under the "Temp/Mass" tab. Finally, she would set the location of the source by going to the "Location" tab. There, she would set the X dimension as "East -> West", indicating that the source should stretch from the east extreme of the mesh to the west extreme. The Y dimension would be set to "North->South". Finally, the Z dimension would be set to "(Value)" and 2.0 would be entered into the edit box. When the solver translation service goes to send this boundary condition to the solver, it will insert the physical locations of the mesh extrema for the X and Y values of the source.

Each boundary has its own C++ class, derived from a parent boundary class. Each time a new boundary is created, its type is instantiated and the new object is inserted dynamically into a tree maintained by the boundary manager dialog object. When the EDIT button is clicked, the boundary edit dialog box is displayed and the current highlighted boundary condition is copied and sent to the boundary edit object (if a folder is selected, the boundary edit dialog box is disabled).

If user has made a change to the boundary condition in the boundary edit dialog box that she wants to keep, she must click "OK" to send those changed back to the master copy in the boundary manager dialog object. Each boundary condition is internally tagged in SCI with a universally unique identifier to ensure that the wrong boundary condition can never be changed.

4.3.2 Mesh Generation

There are two important properties that define a mesh: topology and spacing. An unstructured mesh implies that the elements and nodes do not necessarily have the same topology (number of neighbors), whereas a structured mesh insists on this constraint. A uniform mesh has constant spacing between every node. A

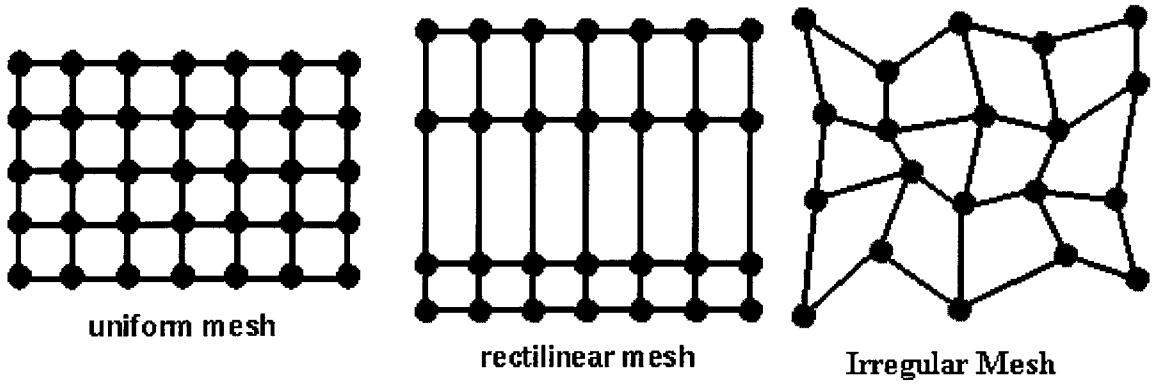


Figure 7: Different Kinds of Mesh

rectilinear mesh allows non-constant spacing between nodes along each axis. An irregular mesh has scattered node spacing throughout the graph.

It is interesting to note the amount of space required to store each type of mesh. The uniform mesh requires only three values – the spacings for each dimension. This is $\Theta(1)$ space in the number of nodes in the mesh. The rectilinear mesh requires the storage of the spacing values between each node on each axis. This occupies $\Theta(\sqrt[3]{n})$ space in the number of nodes in the mesh. The irregular mesh requires the most information, as it requires the location of each node in the graph. This is $\Theta(n)$ in the number of nodes. These amounts become significant when transferring computational meshes on the order of 5,000,000 nodes from SCI to the CFD solver.

The mesh generation tool available in SCI is suited to developing structured, rectilinear three dimensional meshes only. This decision was made for several reasons. One, a rectilinear, structured mesh is sufficient to solve most indoor and outdoor airflow problems. Two, the tool that creates them is simple to use. Three, they require very little storage space.

SCI rectilinear meshes are created through a series of "regions". A region is a collection of nodes along an axis whose spacing is either constant or exponential. The user can specify mesh regions in the mesh generation dialog box.

The user invokes the mesh generation dialog box through a Windows event (selecting it from the menu bar). A Windows event is sent to the document, which calls an edit mesh routine defined in the datalist class (`datalist::editMeshIndex()`). This routine launches the mesh generation dialog box and registers it to receive future Windows events.

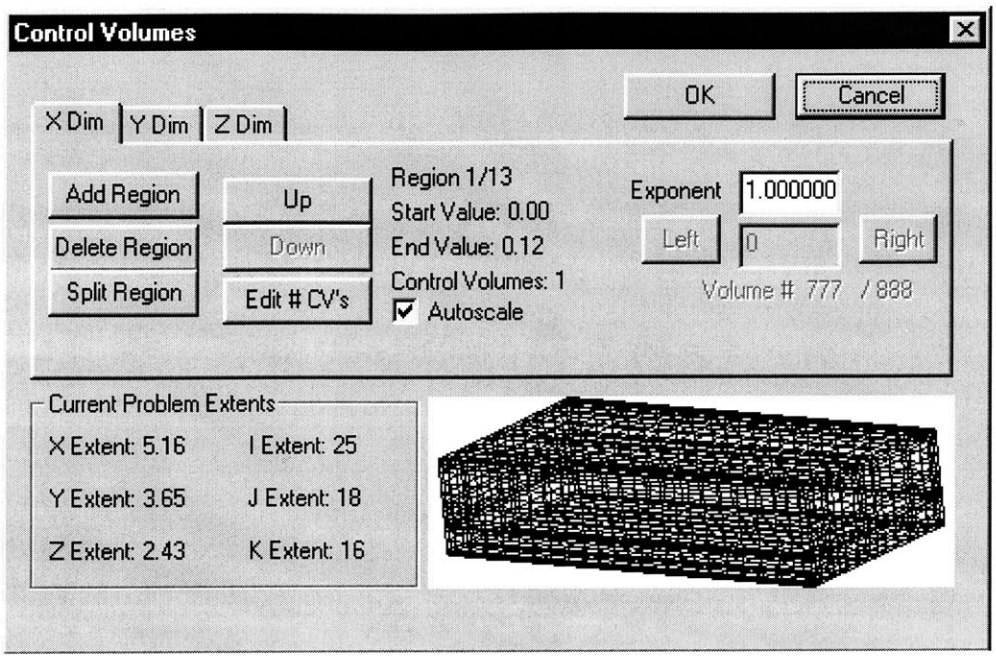


Figure 8: Mesh Generation Dialog Box

The mesh generation dialog box, as seen in the figure, is organized around constructing and editing regions. To add a region along a particular dimension, the user selects the corresponding tab (X Dim, Y Dim, or Z Dim) and then on the "Add Region" button. This causes the launching of the add region dialog box.

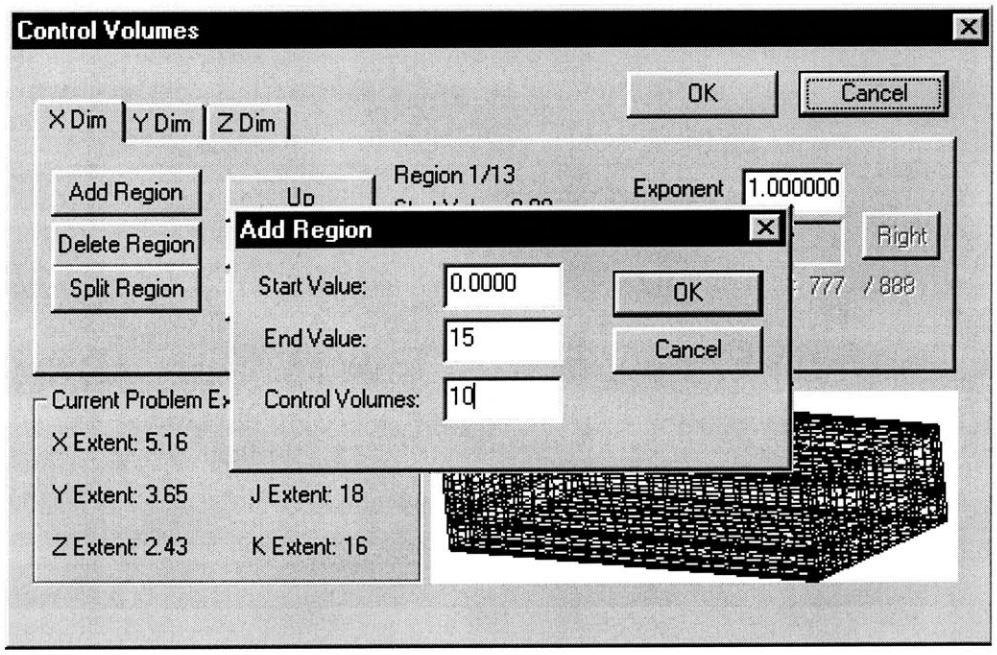


Figure 9: Add Region Dialog Box

If there have been no other regions defined along the selected dimension, then the user is allowed to enter a start location (in meters) and an end location (in meters). The number of control volumes specifies how many zones there will be along this stretch of distance. If there are n control volumes, then there will be $n + 1$ nodes.

Additional regions can only by added to the end of the region list. However, any region can be split into two regions by clicking on the "Split Region.." button, which launches the split region dialog box.

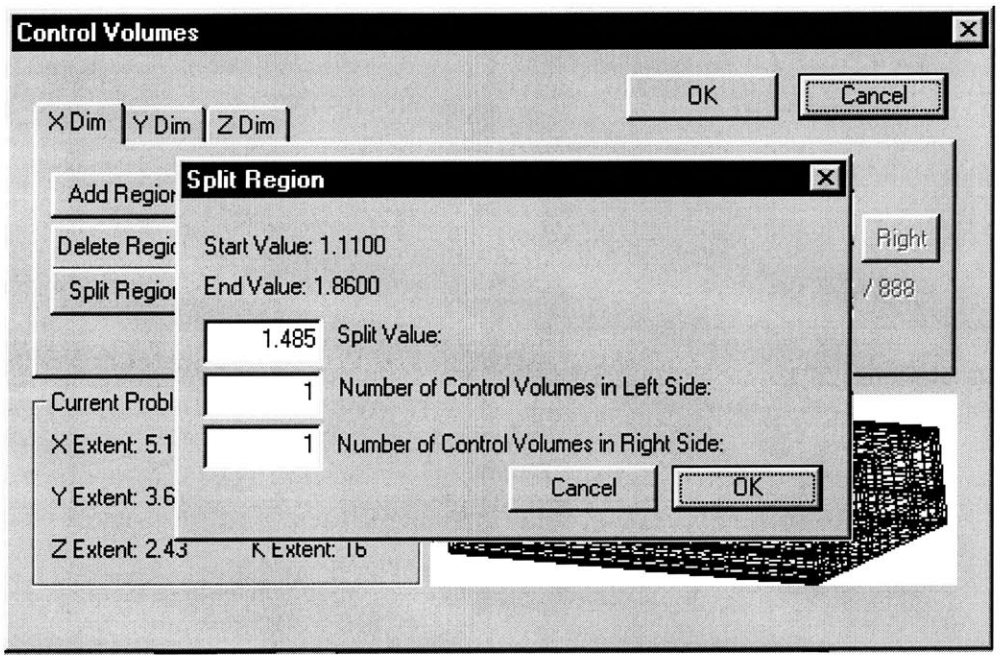


Figure 10: Split Region Dialog Box

Splitting a region allows the user to pick a value in between the start and end values of the region. Two regions are created from the first – one from the start value to the new value, and the other from the new value to the end value.

The mesh generation tool shares coordinate space with the boundary manager. If a nine cubic meter square box, placed at the origin, is entered into the boundary manager, then creating a mesh that runs from a start value of zero and an end value of three (in each dimension) will fully mesh the box.

One of the most common tricks in computational mesh generation is to place more nodes around a region the user feels needs more computational attention. This results in higher accuracy around the denser nodal area. We can achieve this density by decreasing the spacing between nodes exponentially as it approaches the locality.

When the "autoscale" box is checked for a region, the region's control volumes are equally spaced according to the following formula: $d_i = (v_e - v_s)\left(\frac{i}{n_{cv}}\right)^{\exp} + v_s - \sum_{1}^{i-1} d_i$, where d_i is the zonal size (spacing) for zone *i*, v_e is the end value for the region, v_s is the start value, n_{cv} is the number of control volumes in the region, and *exp* is the given exponential growth for the spacing. When the autoscale box is unchecked, the user can enter the spacing for each volume using the edit box on the right hand side of the mesh generation dialog box.

When mesh generation is complete, the datalist object creates an instance of the mesh class. This mesh class in turn creates instances of the region class, one instance for each region defined along each axis. Any SCI classes looking for access to the computational mesh will use the mesh object contained by the datalist object.

4.3.3 Model Parameter Options

Model parameters are specified through three dialog boxes controlled by their corresponding dialog objects. The information gathered by these dialog boxes is stored by the datalist object, which is in turn used by the rest of the system to retrieve this information (primarily, the solver translation services).

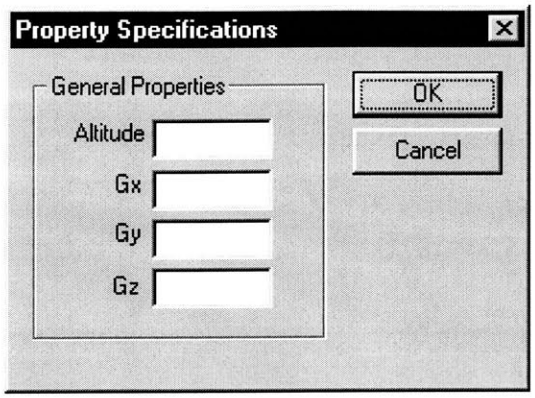


Figure 11: General Properties Dialog Box

The user initiates the data entry process by activating the dialog box. This is done by either clicking on one of the toolbar buttons or by using the drop down menu bar. This causes Windows to fire the appropriate event to the document class, which responds by creating an instance of either the iteration control object, the problem control object, or the model property control object.

Once the user is finished changing model parameters (assuming the user clicked OK and not CANCEL), a reference to the datalist is passed into the dialog object and the changes are copied in. This sets the parameter changes permanently.

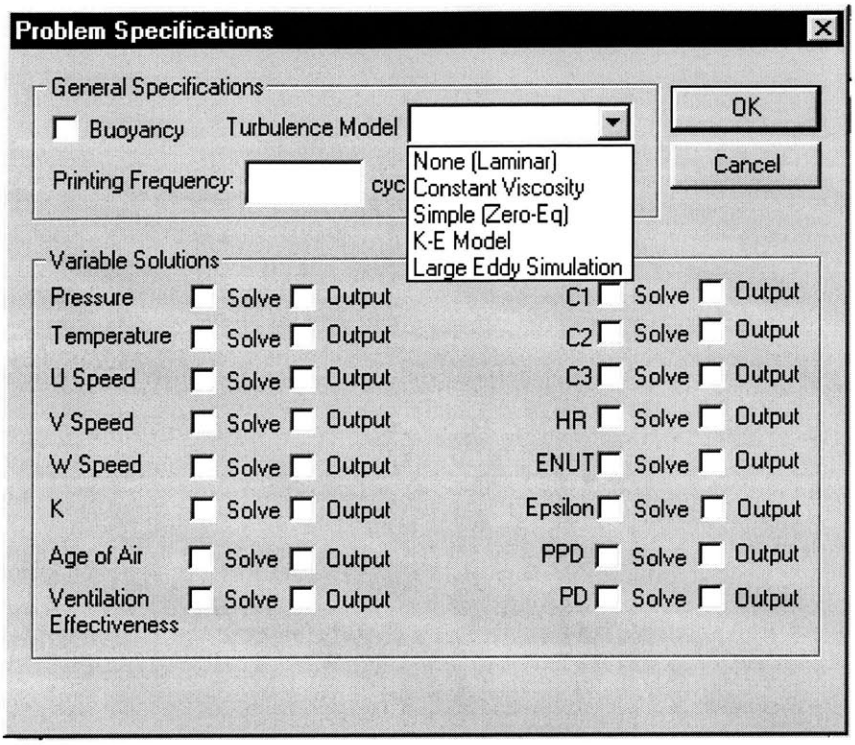


Figure 12: Problem Specifications Dialog Box

The model property dialog box consolidates thirty or forty complex computational parameters into four. Our users are interested in placing buildings on Earth, above ground, and in our atmosphere. This domain restriction implicitly establishes values for most of the CFD parameters (such as beta, air viscosity, conductivity, Prandl and PRT numbers, etc.) The only major factor that can vary is the density of the air, which is directly dependent on the altitude of the building(s).

Occasionally, the geometry of a model might be two dimensional on the XY plane, where "down" is actually in the negative Y direction instead of the negative Z direction. To handle this case, we allow the gravitational constants to be set for each dimension. In most 3D cases, however, Gx, Gy, and Gz will be set

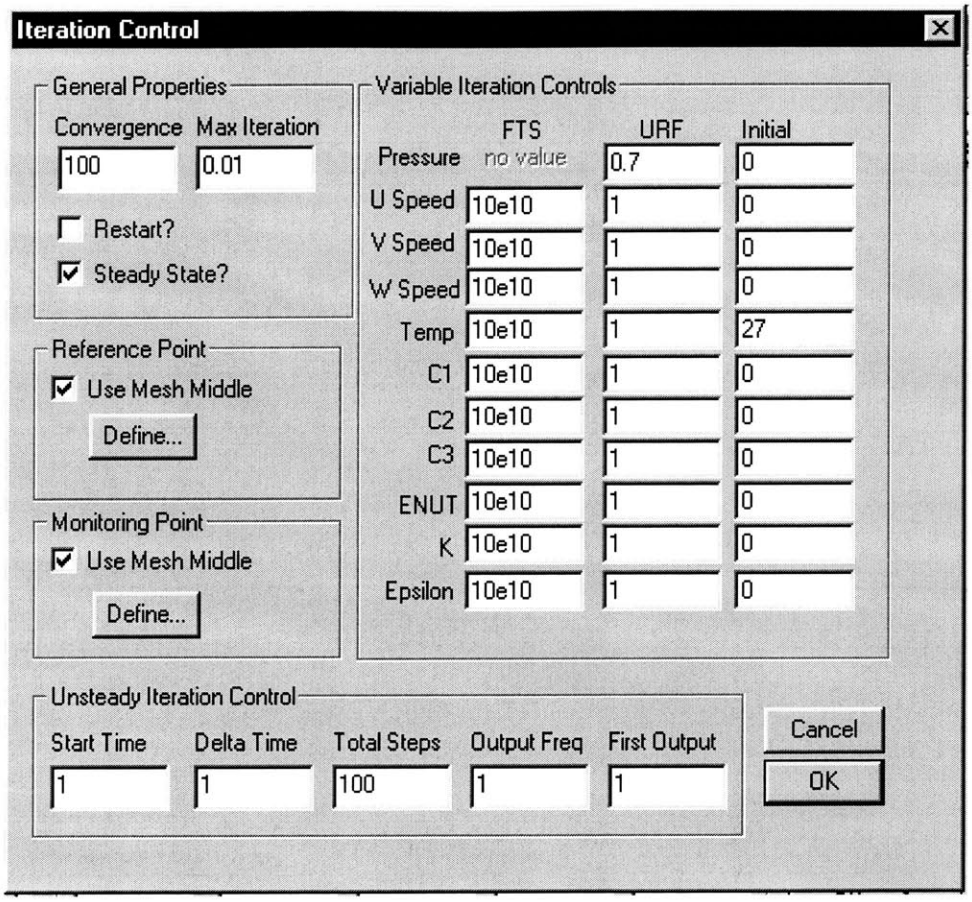


Figure 13: Iteration Control Dialog Box

to 0.0, 0.0, and –9.81, respectively.

The problem-specific dialog box allows the user to specify which values the CFD solver should attempt to solve. It should be noted that these selections are dependent on which solver is selected in the run stage (See Chapter 4.4.1), as not all solvers can solve for all available variables or use all available turbulence models.

The iteration dialog box controls how the solver will approach the solution, how long to iterate for, what the minimum error tolerance is, and how the solver should tune individual values during the simulation.

4.4 Program Infrastructure Classes

The title we give to this class of objects is "Program Infrastructure Objects", but perhaps another deserving name would be "Data Translators", as this describes their primary task. Examine again the GDFD diagram from chapter 4.2. All the objects that bring data in or out of the SCI membrane are included in our list of program infrastructure classes. Their job is to massage data in and out of the SCI framework.

We have mentioned the persistent data storage system often in our discussions about data flow. While all the other systems in SCI map directly to class or a cluster of classes, this system cuts horizontally across all the objects in the system. The persistent storage system is a functional member of each and every object that contains state information. We will discuss how we designed and implemented this system later in this chapter.

4.4.1 Solver Translation Services

A solver translation service incorporates detailed knowledge about a particular CFD solver into the SCI system. When called on to execute, the service gathers the necessary information from SCI's temporary storage objects, formats it, and ships it to the CFD solver. This implies acquiring the geometry and other boundary conditions from the boundary dialog object (via the boundary manager object), obtaining model and iteration parameters values from the datalist object, and the computational mesh from the mesh object (via the datalist object).

Throughout this document, we have left the exact method of calling a CFD solver vague on purpose. Already, we have explored the possibility of using the Internet as a method for running a model. Other

methods are equally as feasible, most of which involve remote procedure calls over some type of data infrastructure.

In our beta version of SCI, we built two solver translation services. One is for a CFD solver called CFD0. CFD0 was written by Ph.D. candidate Jelena Srebric and Professor Qingyan Chen at MIT's Building Technology Laboratory. It distinguishes itself from other solvers through its rapid yet accurate calculation of turbulent flow.

The second translation service was written for a CFD solver called General3D. General3D was written by Ph.D. candidate John Zhai and Professor Qingyan Chen, again at MIT's Building Technology Laboratory. General3D has a wide variety of computational abilities and features, and is the perfect candidate for analyzing how SCI's restrictive interface affect solver utility. General3D can handle irregular, structured meshes of two or three dimensions, and can handle outdoor or indoor airflow. It is written in FORTRAN 90 and runs on all Windows platforms.

The choice of solver is made by the user *after* the entire model has been defined. However, some solvers place special limitations on what kinds of models it can accept. Therefore, it behooves the user to keep in mind which solver she plans on running before reaching the simulation stage.

One of the most important differences to note between the CFD0 and General3D input formats is the specification of boundary conditions. CFD0 does not accept symmetric planes or user-defined sources, whereas General3D does. When the CFD0 translation service comes across a symmetric plane or user-defined source, it disables it in the boundary manager, notifies the user, and skips to the next boundary condition.

CFD0, General3D, and SCI all vary wildly in how they define the location of boundary conditions. CFD0 specifies its boundary conditions by identifying the zones in the computational mesh they occupy. General3D, on the other hand, specifies boundaries by node edges. To make things worse, SCI can (and usually does) defines boundaries freely in the physical domain, independent of the mesh nodes and edges.

This is one of the many perils that befall the solver translation service programmer. How do we convert the locations of boundary conditions? There are only two options. One, we can ask the user how she would like it to be done. Two, we can have the solver translation service take an educated guess based on logical heuristics. It should be clear from our design principles that we will be implementing option two.

To snap boundary conditions to mesh lines, as is required to run General3D, we first look at where each boundary condition is situated with respect to the mesh. If the boundary condition is found to lay outside the mesh, we skip it and move on to the next boundary condition.

If it is within the mesh domain, we need to snap it to mesh edges. One good snapping heuristic is to choose the edge that the boundary is closest to. We need to be careful, however. If the object is a three dimensional box, and it lays entirely inside of a mesh zone, then snapping all the edges to the nearest mesh edge might result in a two dimensional or one dimensional boundary. We would like to maintain the boundary condition's original dimensionality.

Both CFD0 and Genearl3D use the local file system to read their input and dump their results. Once the input file is created by the solver translation service, the solver is executed. The method of execution is left under-specified on purpose, to allow for variation in CFD solvers. For example, one solver could be a Matlab script, while another could be a VB script running under Excel, both of which require different approaches to execute. General3D and CFD0 are executed through the synchronous `_spawnl()` shell command.

4.4.2 STL File Importing

STL (or Stereolithography) are the exported files of computer aided drafting programs. SCI uses them as a way to quickly generate large numbers of geometric boundary conditions (blockages and walls); in other words, to import a geometry.

There is one major problem with STL files: they are comprised entirely of long lists of triangles with no organization, implied or otherwise. Consider the problem of two triangles existing in two-dimensional space, sitting on top of a two-dimensional mesh. It is a simple matter to figure out which zones are contained by these triangles. We can use a top-bottom rasterization algorithm and simple linear interpolation along the bisected edge to fill these boxes.

However, if we move up to three dimensions, a three sided figure no longer encloses a space. This makes deciding which control volumes are blocked and which are not very difficult. Consider a box whose sides have been triangulated. Seen from afar, it is apparent which control volumes are inside the box and which are not. When taken one triangle at a time, however, it is nearly impossible to determine which triangles enclose which spaces.

We sidestep determining which spaces are enclosed by the triangles, as this task is inherently too complicated. Instead, we will turn each control volume that intersects with a triangle into a solid blockage. If the CAD program covered every object surface with triangles, then we should get an enclosed object completely walled up with blockages on every side.

This is not the typical SCI method of defining a geometry. Usually, we define lists of objects located physically in the world and have the solver translation service snap them to the mesh at simulation time. In

this case, we are relying on the mesh to provide us with locations of control volumes that we will turn solid. If the mesh ever changes, then the mesh control volumes need to be recalculated against the STL triangles to determine which ones are blocked and which ones are free.

The STL conversion program works by breaking down the problem into smaller and smaller domains until it is similar to the two-dimensional triangle rasterization algorithm discussed above[3].

The algorithm works on a triangle-by-triangle basis. Each triangle's three points are sorted and the point with the lowest z value is picked as the base point. The lines leading away from the base point to the other two points on the triangle are interpolated to find the locations where they leave the current z-plane of control volumes (if they do not leave the current plane, then the triangle's endpoint locations are used). We will call these two interpolated points the exit points.

The triangle created by the two exit points and the base point are rasterized in the XY plane, marking each block that is within this triangle as filled. If the two exit points were the end points of the triangle, we are finished. If not, the two exit points become the two entry points to the next z level of control volumes. The two triangle sides are again followed until they leave this new z level of control volumes, and two new exit points are found. The quadrilateral formed by the two exit points and the two entry points is triangulated and rasterized. This process continues up the z plane until the triangle is exhausted.

4.4.3 IFC Importing

The organization behind Industry Foundation Class files made importing them into the SCI data framework very simple. The applications programmer who wishes to add IFC file support to his program drops an IFC COM (Component Object Model) server into his code. The server acts as like a snap-in subsystem.

When an IFC file geometry is needed, our application creates an instance of the IFC server class and connects it to the file. Once connected, the file acts like an instantiated object. The program requests a list of geometry objects from the file and uses it to create SCI boundary conditions. The objects are specified as rectangular shapes, which match SCI's geometry primitives, and therefore require no transformation work.

The simplicity of the IFC system suggests that more and more building environment simulation codes will be utilizing it as the substrate of choice for model geometry. Incorporating its use into SCI will be beneficial to the system for years to come.

4.4.4 Persistent Object Storage

The one system that does not occupy its own class of objets is the persistent object storage system. The purpose of persistent storage is to save and load SCI documents to a permanent storage facility. Storage is considered successful if a document load operation restores all visualizations, results, and parameters as if the application had never left it.

Not all data in the application qualifies to be recorded by the persistent storage system. For example, SCI keeps track of the file path location of the last CFD solver used, through one of the document class's data members. The product created by the persistent storage system is highly portable and could end up on several different computers, most likely rendering a file path incorrect. Therefore, it makes little sense to record this particular and transient information.

---

[3] Note that the following algorithm requires uniform mesh spacing, which is a special case of the SCI mesh generation tool.

However, most other object data does require persistent storage. MFC applications, including SCI, typically implement the storage system using the `serialize()` paradigm. Serialization, a verb indicating the transformation of object data to a stream of bits, is accomplished using MFC's CArchive data structure. When the user indicates she wishes a permanent record of her data be created (usually through the "Save" command under the file menu), a CArchive object is instantiated and sent to the document object's `serialize()` method.

The CArchive object that arrives at the document's `::serialize()` method has already been prepared to send or retrieve data to/from a file chosen by the user through a file dialog box. The method merely has to pipe data in or out of the object, depending on whether or not the user is saving or loading, using CArchive's overloaded piping operators (>> and <<). These operators are only defined for basic data types, such as integers and floats. For member data that is not a basic data type (all SCI classes fit this category), serialization is done by calling the object's `::serialize()` function. Thus we can see that each class in the SCI system needs to have this member function specified and implemented.

Each class's `::serialize()` method should perform the requested save or load operation by piping its data members to the archive and calling `::serialize()` on any member objects.

Consider the problem of multiple objects containing references to the same object as a data member. If we followed the implementation of `::serialize()` calls as specified above, the referred-to object would find itself serialized more than once, uselessly duplicating its data in the storage system.

This can be a serious problem. For example, many parts of the SCI system use the datalist object to access results from the CFD solvers. This object could contain data points for several variables defined over a computational mesh representing over one millions nodes. Each storage hit might result a data increase in the storage system of over five or six million `double` values.

SCI solves this problem by assigning each instantiated object a "custodian". While serializing, an object only serializes member data if it is has "custody" over it. If it does not, then that member data is skipped, for it will be stored by the object which does have custody. Every object in SCI always has one and only one custodian who will always be reached during the serialization call path. This constraint is maintained through program design and incurs no operational overhead.

If no permanent custodian for an object could be found, then SCI would move to a system whereby the document would obtain a universal unique identifier at the start of serialization. This number would be passed along with the serialization requests. Objects would check the identifier against one they already hold. If the numbers differ, the serialization would be performed and the object would keep a copy of the identifier. If it was equal, then the object would perform no serialization, since this would imply that the object had been already reached in the call path. This algorithm is similar to the operation of mark and sweep garbage collection routines.

The unique identifier algorithm incurs a very minimal, but non negative, amount of operational overhead and space consumption. It is a moot point, however, since all SCI objects have permanent custodians and we do not need implement it.

4.4.5 Security

The SCI system is currently used by hundreds of graduate students at MIT. This has put the code into minor distribution, appearing in pockets of industry and a few educational institutions. As the developers of SCI, we would like to keep track our system's migration. We do this for a number of reasons. One, we want to maintain a two-way channel to communicate bugs and suggestions to our users. Two, we want to provide our users with updates and product suggestions that they might be interested in. Three, we want to

engage CFD programmers who might find SCI useful and invite them to write solver translation services for their codes.

All these benefits could be realized through a voluntary registration feature. However, a fourth benefit can only be achieved through security: protection against theft of service. At some point, SCI might be packaged as part of marketed product. When this happens, we would like the code to be available only to users who have offered compensation.

It is useful to think about a security scheme as a knowledge control system. The optimal security system is a method by which only authorized users have the knowledge necessary to open the application. This knowledge can only be granted by a central system, which is under the developer's control. Furthermore, replication of the activation knowledge to unauthorized users should ideally render the knowledge useless.

This is not as impossible as it sounds. A security system known as RSA (Rivest-Shamir-Adleman), invented at MIT, encrypts and decrypts data using complementary pairs of numeric keys. For example, if one "key" is used to encrypt the data, then the other "key" is used to decrypt the data. The mathematics behind RSA make it very difficult to calculate one key from the other key.

A scheme known as public key encryption uses RSA key pairs for message encryption and authentication. In the process, one key from the pair is released to the public, and the other is kept private. When data is encrypted by the author using the private key, users decrypt it with the public key. Users can be confident that the message is in fact generated by the author, because only the author would be able to encrypt data with the private key.

Our system will make use of RSA by assigning the SCI application a public/private key pair. When the program is launched, it reads the volume information number off of the main hard drive. The volume

information number is a random integer written to the drive each time it is formatted by Windows. The user is asked to send this number in to the central server. We will call this message m_1.

The developer takes m_1 and encrypts it with SCI's private key to create $m_2 = k_{priv}(m_1)$. This is sent back to the user, who enters it into the application. SCI then attempts to decrypt m_2 using the public key that it was shipped with. If $k_{pub}(k_{priv}(m_1))$ [or, $k_{pub}(m_2)$] matches the volume information number on the hard drive, then the application is launched. m_2 is recorded in the Windows registry, so that the next time SCI is launched it can get a copy of m_2 without bothering the user or the developer.

If we make the assumption that each computer in the world has a different volume information number, then this system is infallible, so long as SCI's private key remains private. Pretend that m_2 was sent to a different computer. Decrypting m_2 would result in a different volume information number, and therefore the program would not launch.

Unfortunately, this system is limited to the security of the volume information number. There exist programs that can change this number at will. One way we can contain this flaw is to include an expiration date into m_2. This would require unauthorized (and authorized) users to reacquire a new m_2 each time the expiration date lapses. This creates a hassle for the authorized user, but an even bigger hassle for the unauthorized user. Unfortunately, there are also programs which can fool an application into believing it is any date the user desires. Chip manufacturer Intel has recently considered hard wiring a unique identifier onto each of their processors. Should this become a standard practice, the processor number could replace the volume information number and eliminate the flaw.

The RSA system is patented under United States Patent #4405829. However, this patent expires in the latter half of 2000. Therefore, the RSA security section of SCI will remain unimplemented until the patent protection lapses. In its place, the cipher-block system Data Encryption Standard (DES) is used to encrypt m_2. This system uses only one key, so we are forced to distribute the single decryption and encryption key along with SCI. In addition to the other attacks mentioned above, this system can also be attacked by examining the SCI binary, locating the DES key, and creating a m_2 without contacting the central service.

4.5 Visualization Classes

The data returned by the CFD solver, the model geometry, and the computational mesh are all displayed on the monitor by the visualization subsystem. The user interface side of the subsystem allows the user to specify which results to view and how to view it. The graphics core side of the subsystem accumulates data from other parts of SCI and generates the OpenGL rendering commands.

The main window of SCI is the drawing pad for the visualization classes. Initially, the top right corner of the window is occupied by the boundary manager dialog box, and the bottom left corner is occupied by the plot manager dialog box. SCI draws a mini 3-D axis in the bottom right corner of the window so the user can keep her bearings during three dimensional rotations. But the majority of the window, the center, is left blank. This space is reserved for data visualization.

4.5.1 Visualization User Interface Classes

The GDFD presented in chapter 4.2 shows the user interacting an object called the "plot manager". The plot manager is the user's front end to the visualization classes. The plot manager controls an object called the plot manager dialog box in the same relationship that the boundary manager and the boundary manager dialog box share (see chapter 4.3.1). The plot manager launches the plot manager dialog object, which in turn handles the Windows events.

The plot manager dialog box allows users to select new plots to display in the center of the big window. There are six different kinds of plots offered to the user: vector, pseudocolor, contour, boundary, mesh, and STL. Each plot type has its own C++ class of the same name, and all these classes are all derived from an abstract parent class called `plot`.

When a plot is created, an object of its type is instantiated and inserted into a list managed by the plot dialog object. This occurs when the user clicks on the New Plot dropdown list, located above the DEL button. The list of currently instantiated plots is displayed in the leftmost combo box. When the user selects one of these plots, its attribute controls appear to the right of the plot list. These controls are used to modify the parameters for that particular plot, such as color, variable, line thickness, etc. The types and parameters managed by the controls on the right side of the window depend on the plot type selected on the left.

For example, a pseudocolor plot (which is a hot-to-cold shading of a scalar data field) is selected in figure.

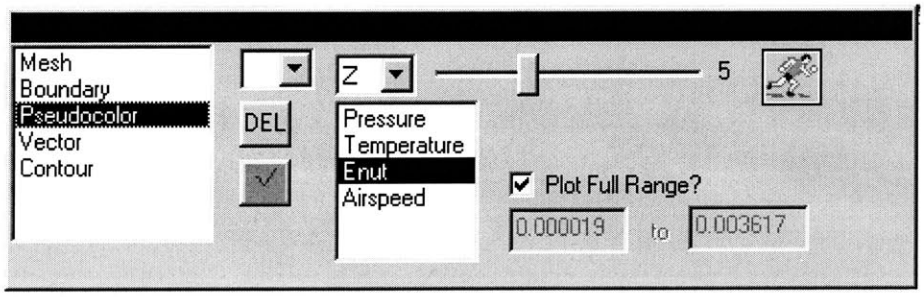


Figure 14: Plot Manager Dialog Box

The green checkmark below the DEL button indicates that the plot is active. Active plots are displayed in the main window. Inactive plots are not rendered to the window. Marking a plot inactive is simpler and takes a little less time than deleting it, should the user feel she might want to reactive the plot later.

The dropdown and slider on the top of the plot manager function to identify the data "slice" to plot. The pseudocolor, contour, and vector plots operate only on two dimensional slices of three dimensional data. The dropdown control allows the user to select the dimension to slice on, and the slider picks the logical node in the computational mesh to use. In the figure, the "Z" dimension is selected on the dropdown and node 5 is selected on the slider. This means that data on the $z = 5$ plane will be visualized. Note that this does not refer to five meters, but rather to the fifth logical node in the computational mesh (CFD programmers refer to the mesh's logical axis as i-j-k instead of x-y-z, but we chose to use x-y-z in the dropdown to avoid user confusion).

Clicking on the "running man" icon animates the slider through all the nodes along the dimension selected.

The combo box in the middle of the plot manager dialog window allows the user to select the variable she wishes to plot. In figure, the variable "Enut" is selected. Only variables that have been solved by the CFD solver and subsequently stored in the datalist object are available for selection.

By default, the variable's minimum and maximum values correspond to the ends of the hot-to-cold coloring in a scalar data pseudocolor plot. However, the user might want to change these boundary values to something more inclusive if one or two values in the variable are throwing the extreme values off. To specify new maximums and minimums, the user unchecks the "Plot Full Range?" button and enters new values in the edit boxes.

For a contour plot, the controls on the right side of the plot manager dialog box look similar to a pseudocolor plot except there is an extra slider. This slider corresponds to the number of contour divisions to draw. All other controls are similar in function to the pseudocolor controls.

The vector plot controls have the same slice selection controls as the pseudocolor and contour plots, but do not allow the user to select a minimum and maximum value, for they have little value in reference to vector data. When a vector plot is selected on the left, the slider beneath the running man icon corresponds to the tail size of the vectors on the window. A small value on this slider results in a small vector tail, and a long value results in a longer tail.

Mesh plots also have the slice selection tools, and have a color dropdown and a checkbox marked "Boundary only?". Checking the box results in only the mesh boundaries displayed. Leaving it unchecked results in drawing each volume in the computational mesh. The slider on the right varies the thickness of the lines used to draw the mesh.

The boundary plot has two controls on the right side of the plot manager dialog box: a color selector and a line thickness slider. It also has a box with a replica of the boundary hierarchy from the boundary manager. When the user selects a boundary, the boundary's borders are plotted in the main window. If the user selects a folder, then all the boundaries found inside that folder are drawn.

Any time a change is made to one of the plot controls, the dialog manager fires a redraw message to the application using Windows' message passing infrastructure. The redraw message eventually finds its way to the plot manager, who examines each plot in its list. Each active plot is then allowed to render itself by calling its `::paint()` method. We will discuss the `::paint()` method in the following chapter.

The eyeball object keeps track of how the user is viewing the model. Think about the model's plots as existing somewhere in space. The eyeball is a point in the model's coordinate system, "looking" at the center of the model. The eyeball can be translated and rotated around the model. Wherever the eyeball "sees" is what is painted in the big window.

To make the user feel like she is controlling the eyeball, mouse movements are tracked by the Windows system and sent to the eyeball so it can update its position. If the mouse is dragged (moving the mouse while the left button is clicked), the eyeball registers a rotation around the axis defined by the line perpendicular to both the line extending from the eyeball's current location to the center of the model and the line drawn by the mouse movement. If the control key is held down, the eyeball is translated in two dimensions along the plane defined by the perpendicular bisector of the line between the eyeball and the center of the model. A right mouse button click moves the eyeball 25% closer to the center of the model, and a control-right click increase the distance by 125%.

4.5.2 Visualization Core Graphic System

Each plot class is derived from the abstract class `plot`. This abstract class specifies the type signature for three virtual functions that each class must implement: render (datalist *dl, slice *s), legend (datalist *dl, int pos), and the serialize (CArchive &ar). The first two methods produce OpenGL commands from SCI data. The last method implements persistent storage for the plots (see chapter 4.4.4)

OpenGL is the most widely used and supported 2D and 3D graphics API in the computer industry. It allows an application to specify geometric objects through its vertex primitives, and then specify how to render these objects into a framebuffer. Once rendered, the framebuffer is copied to the window through a Windows bitblt routine. For a background on rendering theory and terminology, consult the authoritative work *Computer Graphics: Principles and Practice, Second Edition* by Foley and Van Dam.

OpenGL keeps a running list of rendering options in memory, such as current line thickness, color, and the current rotation and mapping matrices used to render three-dimensional objects onto a two-dimensional window. It executes commands as they are passed to the engine. Commands either modify the current rendering options or specify a geometric object to draw.

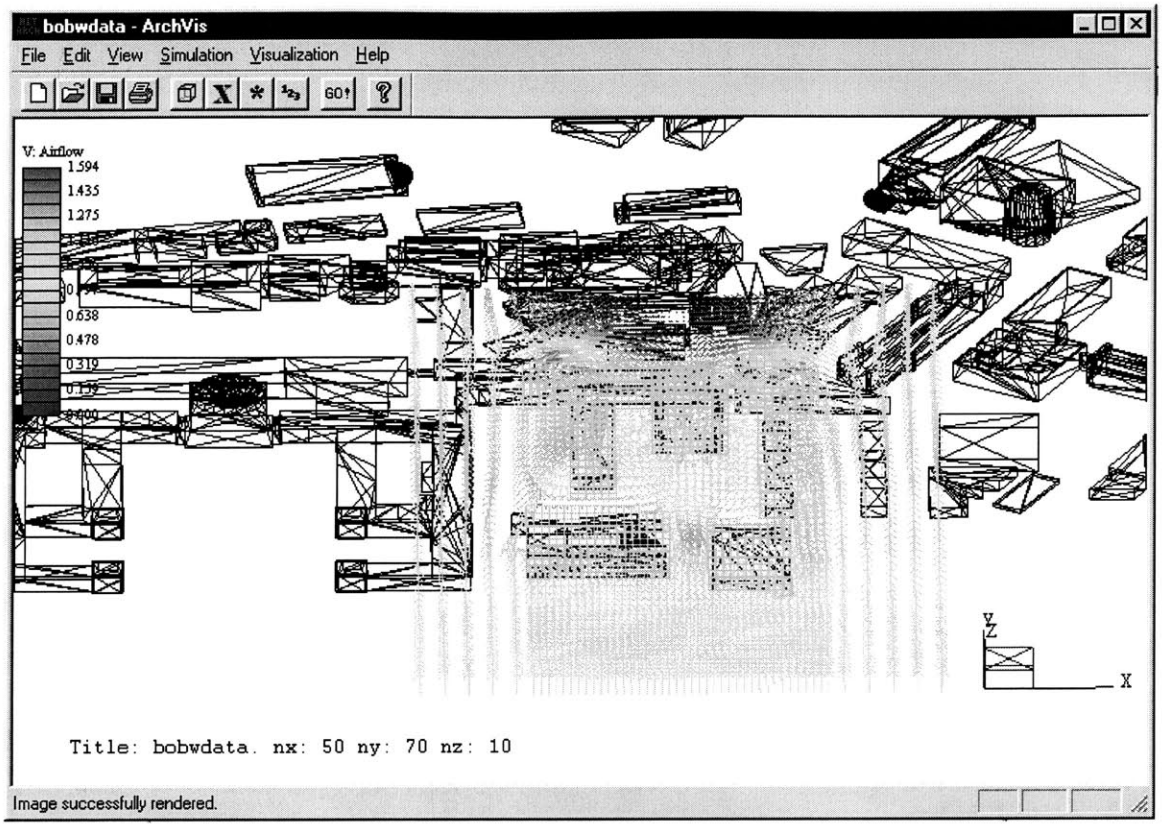


Figure 15: This map of MIT campus was imported through the STL file importing object. The Green Building, in the middle, was converted to boundary conditions via the STL boundary condition conversion algorithm. The model was solved using General 3D. The vector flow shown is airflow at the ground level. Notice the strong air current flowing in between Walker Memorial and Hayden Library.

The plot dialog object has responsibility for aggregating the OpenGL commands to build the model. This is accomplished in two steps: first, the OpenGL environment is initialized, and second, the `paint()` method for each plot object in the active list is called.

To initialize the environment, OpenGL's framebuffer is cleared. At this point, the eyeball object is consulted for its position in the model space (if the model geometry has recently been generated, the eyeball chooses a default starting position based on the mesh extremes stored in the datalist). This information is used to construct the rotation and mapping matrices for a parallel projection.

The rendering methods for the plot objects turn SCI data into OpenGL commands. We will examine each plot type's render function individually, starting with the simplest and moving to more complicated constructions.

Figure 16: A pseudocolor plot of the airspeed in a room. The high speed towards the right is the result of an inlet on the left hand wall.

When a stereolithography file is imported into the system, two things take place. First, the triangles are turned into boundary conditions using the algorithm presented in chapter 4.4.2. Second, the triangles are stored in the datalist object. There are two ways for the user to look at the stereolithography information – either the user can plot the boundary conditions generated by the triangles, or the user can look at the triangles themselves. When the user wants to look at the raw triangles, she requests an STL plot object.

The STL plot object has a very simple rendering method. First, it changes OpenGL's working color to the color the user requested in the plot controls (see chapter 4.5.1). Second, it steps through each of the n

triangles in the file and sends them to OpenGL. OpenGL has a number of graphics primitives, and one of them happens to be a triangle.

To enter a triangle into the model, the plot object issued the "triangle" command, glBegin (GL_TRIANGLES). This prepares OpenGL to receive triangle data. Then, it issues n sets of three vertex commands (glVertex (x, y, z)) to specify the location of each vertex of the triangles. After the vertex are entered, OpenGL is told that there are no more triangles coming, (glEnd ()), and the rendering is complete.

SCI's computational mesh is stored in the datalist object as a collection of regions. To render the mesh into OpenGL primitives, the location of each node must be computed. This is calculated from the formulas $v_{x_{i,r}} = \sum_{1}^{i} d_i + x_{\min_r}$, $v_{y_{i,r}} = \sum_{1}^{i} d_i + y_{\min_r}$, $v_{z_{i,r}} = \sum_{1}^{i} d_i + z_{\min_r}$, where $v_{x_{i,r}}$ represents the x component of the i^{th} node in region r, d represents the spacing for control volume I, and $x_{\min_r}$ is the x coordinate where the r^{th} region begins. The same form follows for the y and z components of the node.

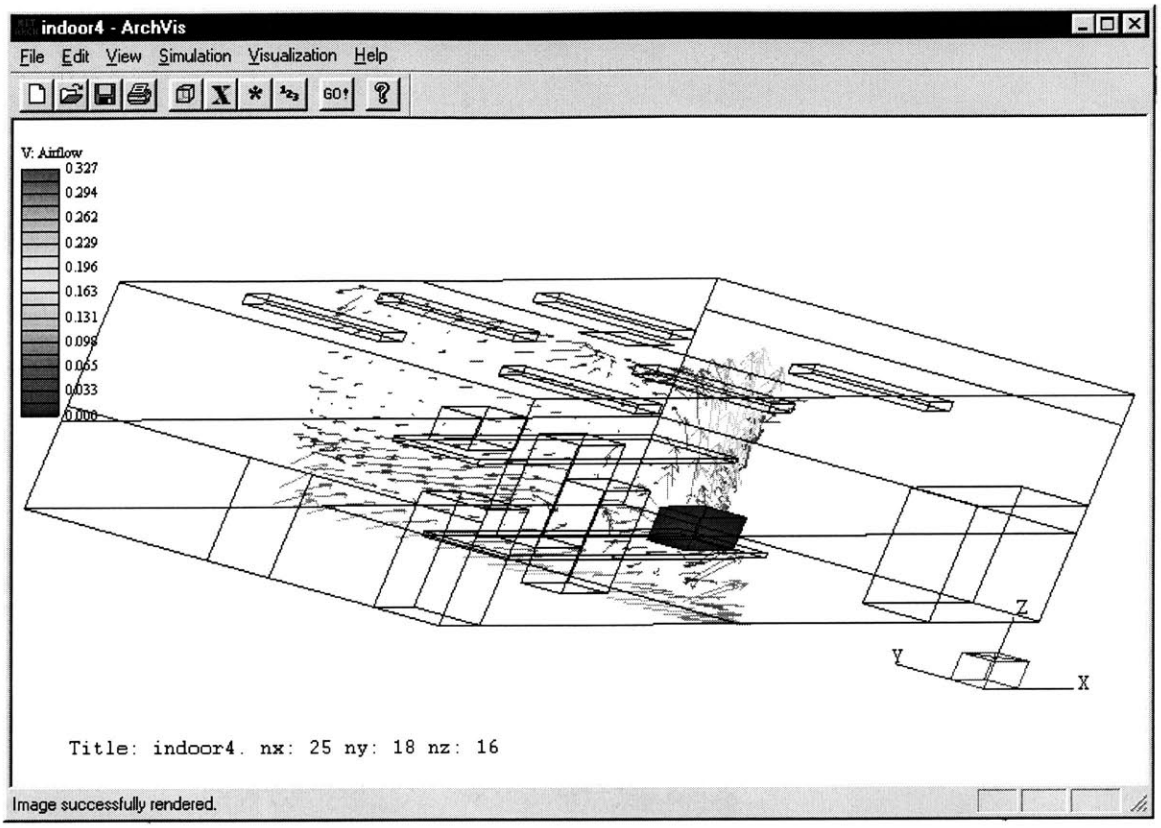


Figure 17: An indoor airflow model, solved with CFD0. The solid box is a warm computer. The vector plot is of the air vectors. Notice the convection above the computer.

Instead of triangles, the mesh render method uses an OpenGL primitive called a quadrilateral strip (glBegin (GL_QUAD_STRIP)). A quadrilateral strip connects vertex in the fashion indicated by figure. In this way, the XY planes of the mesh are drawn, one z-plane at a time. Then, to fully connect the mesh, the YZ planes are drawn, one x-plane at a time.

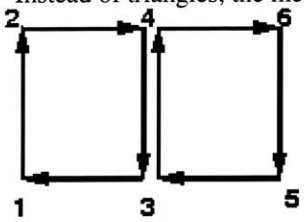


Figure 18: GL_QUAD_STRIPS Vertex Order

The boundary render method only has to draw a box or a plane for each boundary condition. Both are rendered in exactly the same way as the mesh, using OpenGL's quadrilateral strip primitive.

If OpenGL is placed in GL_FILL mode, then closed polygons such as the quadrilateral strips are filled in with color. The fill color depends on the colors associated with each vertex of the square. If each vertex has

a different color, then the color inside the square are linearly interpolated. How this linear interpolation is done depends on the particular implementation of OpenGL, but usually the polygon is triangulated and color data is linearly interpolated along triangle edges.

We use the GL_FILL mode to render our pseudocolor plots. A single plane of the mesh is entered into OpenGL the same fashion as one of the mesh planes is entered, using GL_QUAD_STRIP, except this time, a color based on the value of the scalar variable is associated with each vertex. The scalar data is obtained from the datalist object that is passed in to the render method.

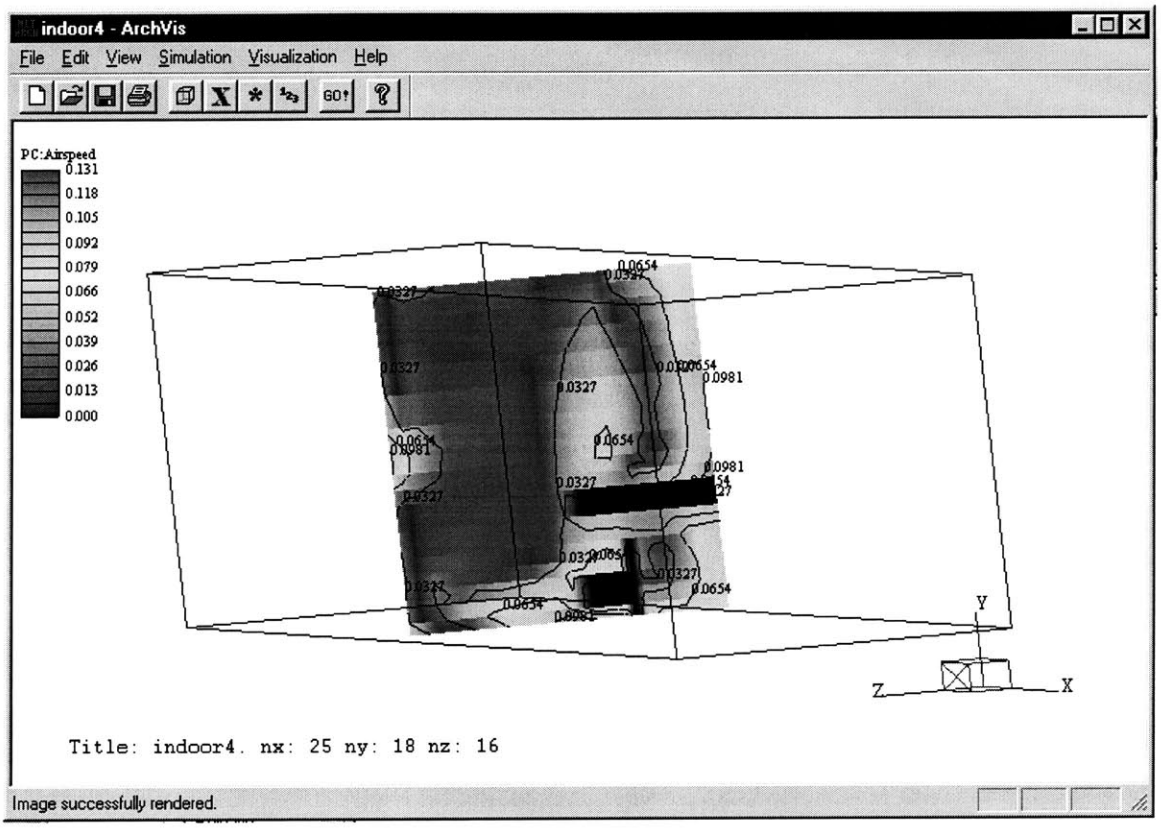


Figure 19: A contour plot of the airspeed of an indoor airflow model. The airspeed is also pseudocolored. Notice that the contour lines trace lines of similar color.

To compute the color of a given vertex, the render routine first normalizes the data value at that node to a value from [0-1). This is done by subtracting the minimum data value and dividing by the maximum, as shown in the formula. The minimum and maximum values are selected from the entire data set for the selected variable. If the user has specified her own minimum and maximum values by unselecting "Plot

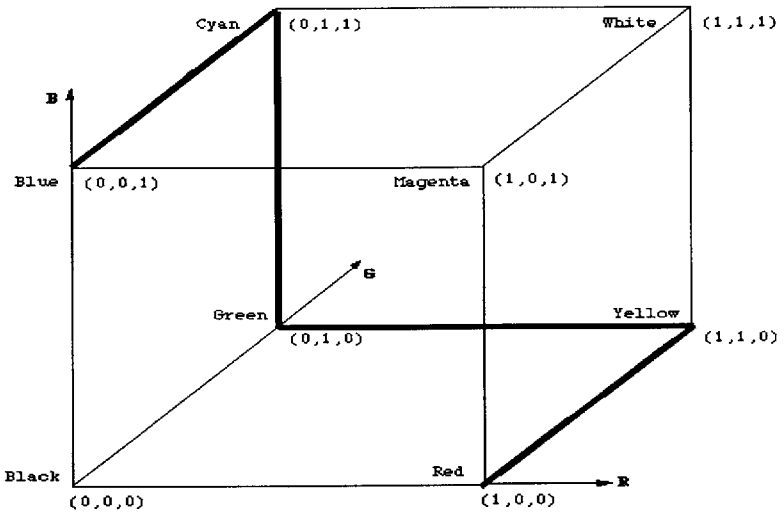


Figure 20: The color cube used to calculate the RGB hot-cold colors. The bold line is followed and 255 RGB sample values are taken at regular intervals.

Full Range?" in the plot manager dialog box, then those values are used instead of the true minimum and maximum (assuming the maximum is not zero).

$$d_{0,1} = \frac{d - d_{\min}}{d_{\max}}$$

$d_{0,1}$ is used to index into an array of RGB colors, sorted from hot to color. These RGB values are calculated by interpolating along a path in a color cube. See figure. The index value is calculated by multiplying $d_{0,1}$ by the number of entries in the RGB color array and taking the integral floor.

Vector data associates vectors (magnitude and direction pairs) with points on the computational mesh. Air velocity is an example of vector data. It is useful to draw arrows whose magnitude and direction represents the vector data. SCI's vector plot draws fields of these arrows to visualize how the data is flowing around the mesh.

A quick way to display this data using OpenGL is to set the rendering machine into line segment mode (GL_LINES). In this mode, pairs of vertex sent to the render engine are connected by a simple line. Given

that we have the locations of the nodes $\{n_x, n_y, n_z\}$, and the vector data $\{v_x, v_y, v_z\}$, we can easily send the vertex $\{n_x, n_y, n_z\}$ for the node location, and $\{n_x + v_x, n_y + v_y, n_z + v_z\}$ for the vertex data endpoint.

However, there is no assurance that the vector data and the mesh are based on the same coordinate system. It could be the case where the node distances are three or four orders of magnitude larger than the vector data, rendering our vector lines too small to see. There are two solutions to solve this problem: one, we can guarantee that the average vector line takes up a given percentage of the screen, or two, we can guarantee that the average vector takes up a given percentage of the mesh size.

Option one would result in vector lines remaining the same size no matter how closely the eyeball zoomed in and out of the mesh, since the vector length would be based on how large the window is. Option two would result in the vectors increasing in size commensurate with the mesh. We chose to implement option two. Option one looks awkward, and requires the viewing matrices be inverted every time the user changes the eyeball location. This is computationally expensive and results in noticeable slowdowns in render times. Option two requires little computational overhead and only needs to be performed once.

In addition to a simple line indicating where the vector is pointing, we would like to place an arrowhead at the tip of the vector line. This aids in visualization of the data flow. Since our vector data points in three dimensions, we would like three arrowhead lines extending from the tip of the main vector line so we can see them no matter what angle we are looking. The arrowhead lines should lift 26.5 degrees off of the vector line, and extend ¼ of the length back down the vector. The three lines should be separated from each other by 120 degrees in the vector line's perpendicular plane.

These requirements are best understood if we examine a simple vertex line. We will use the line that begins at the origin and goes to $\{0,0,1\}$. The three arrowhead lines will each begin at $\{0,0,1\}$ and end at three different points in space. Since the endpoints of the arrowheads come down to ¼ of the vector line, this means that all the endpoints have $z = 0.75$. If this is true, then to achieve a lift of 26.5 degrees from the vector line, we need to satisfy the equation $\tan 26.5 = \frac{x}{0.25}$, which makes $x = 0.125$. Thus, the three endpoints for the arrowheads lie equidistant on a circle (to achieve 120 degree separation) of radius 0.125 on the $z = 0.75$ plane. Three points which satisfy these requirements are $\{0.108,0.0625,0.75\}$, $\{-0.108,0.0625,0.75\}$, and $\{0,-0.125,0.75\}$.

These points are only valid for the vector that starts at the origin and goes to $\{0,0,1\}$. We will use these results by figuring out how to turn every vector into this simple case. Then, we will apply the inverse of this procedure to our three satisfactory points to get the points we are really after.

To translate a vector $\{v_x, v_y, v_z\}$ to $\{0,0,1\}$, we need to first rotate it about the Y axis by θ_y degrees until we zero the Y component, leaving us with $\{v_{x-roty}, 0, v_{z-roty}\}$. Then, we will need to rotate this vector about the X axis by θ_x degrees until we zero out the X component, resulting in $\{0,0,v_{z-roty-rotx}\}$. Then we will normalize this vector by a factor s to result in $\{0,0,1\}$.

We can calculate the unknown angles and scale factors by the following equations:

$$\theta_y = \tan^{-1}\left(\frac{v_x}{v_z}\right)$$

$$\theta_x = \tan^{-1}\left(\frac{v_y}{\sqrt{v_x^2 + v_z^2}}\right)$$

$$s = 1/v_{z-roty-rotx}$$

The inverse of this process is to divide by the scale factor and redo the rotations in the reverse order and using the negative angles. Thus, we compute the following matrix multiplications:

$$s^{-1}[-0.108 \quad 0.0625 \quad 0.75]\begin{bmatrix}1 & 0 & 0\\ 0 & \cos(-\theta_x) & -\sin(-\theta_x)\\ 0 & \sin(-\theta_x) & \cos(-\theta_x)\end{bmatrix}\begin{bmatrix}\cos(-\theta_y) & 0 & -\sin(-\theta_y)\\ 0 & 1 & 0\\ \sin(-\theta_y) & 0 & \cos(-\theta_y)\end{bmatrix}$$

$$s^{-1}[0 \quad -0.125 \quad 0.75]\begin{bmatrix}1 & 0 & 0\\ 0 & \cos(-\theta_x) & -\sin(-\theta_x)\\ 0 & \sin(-\theta_x) & \cos(-\theta_x)\end{bmatrix}\begin{bmatrix}\cos(-\theta_y) & 0 & -\sin(-\theta_y)\\ 0 & 1 & 0\\ \sin(-\theta_y) & 0 & \cos(-\theta_y)\end{bmatrix}$$

$$s^{-1}[0.108 \quad 0.0625 \quad 0.75]\begin{bmatrix}1 & 0 & 0\\ 0 & \cos(-\theta_x) & -\sin(-\theta_x)\\ 0 & \sin(-\theta_x) & \cos(-\theta_x)\end{bmatrix}\begin{bmatrix}\cos(-\theta_y) & 0 & -\sin(-\theta_y)\\ 0 & 1 & 0\\ \sin(-\theta_y) & 0 & \cos(-\theta_y)\end{bmatrix}$$

These three equations will result in endpoints that, once translated up to the node by adding $\{n_x, n_y, n_z\}$, can be connected with the tip of the vector line $\{n_x + v_x, n_y + v_y, n_z + v_z\}$ to result in the three arrowhead lines we desire.

A contour plot sketches lines of representing constant value in two-dimensional scalar data fields. Calculating the line segments which make up these contours lines can be decomposed into drawing the lines inside a single zone for each zone.

Since data is only available at the four corners of the zone, we will need to triangulate the square by dividing it up along one of the two diagonals. This creates two triangles. Using these triangles we can decompose the problem into drawing contour lines inside of triangles whose data values are known along their entire border through linear interpolation.

Creating contour lines inside of triangles is simple. We think about the three points of a triangle as having two sets of data: one, the location in x, y, and z space, and two, the scalar data value of the variable at that point. We start by sorting the points by their scalar data value component, and choosing the high and the low points. These two points are connected in the triangle by a physical line stretching between their locations. We will call this line our major triangle side, and the other two lines our minor triangle sides.

We look to see if any scalar data values on the major line are one of the contour values we are plotting. Since we are doing a linear interpolation, this simplifies down to seeing if one of the contour values is contained between the scalar data values of the two major line points. If there is, then we find out how far down the major line the point is. This will give us the x, y, and z coordinates of the point.

Necessarily, if the contour line enters the triangle at this point, then it must leave the triangle through one of the minor sides. We calculate the coordinates of the exit point through linear interpolation, and connect the two points by a line segment. Repeating this process for all the contour values in all the triangles in all the zones will result in the contour map we are looking for. See the appendix for the OpenGL rendering code for this algorithm.

5. Final Thoughts

The SCI design and implementation work is a success story. The system has been in use by new students in the field of environment simulation at MIT for a year now. Within hours of introduction to the software, users are able to generate a simple building geometry and run a CFD computation. The increased comfort that comes with these early successes furthers the new user to investigate more complex and advanced uses of the program. This breeds exactly the kind of lock-in we were hoping to achieve with SCI.

Regardless of these successes, there remains significant work to be done. We would like to become larger participants in the international consolidation of the building environment simulation community. We expect that the industry will continue to approach an application that complete in a single interface all the analysis needs of an architect: lighting, energy, airflow, virtual reality visualization, and others. Our ideas on how to consolidate many airflow codes together under a single interface, and on how to prevent the user from being overwhelmed by computational details, will have a positive impact on this movement.

6. Bibliography

[1] Srebric, J., Chen, Q., and Glicksman, L.R. "A coupled airflow-and- energy simulation program for indoor thermal environment studies", Accepted by ASHRAE Transactions.

[2] Chen, Q. and Kooi, J. van der. 1988. "ACCURACY - a computer program for combined problems of energy analysis, indoor airflow and air quality," ASHRAE Transactions, 94(2), 196-214.

[3] Broderick III, Charles R., 1997-1999, "MITFlow: CFD Interface", unpublished.

[5] Microsoft First Class Frequently Asked Questions. Microsoft Corporation. 15 May 1997. <http://msdn.microsoft.com/library/backgrnd/html/msdn_mfcfaq50.htm>.

[6] OpenGL Specification v. 1.1. Silicon Graphics Inc. 1997. <http://trant.sgi.com/opengl/docs/Specs/glspec1.1/glspec.html>

[7] Stereolithography Data Format Specification. Ennex Corporation., <http://www.Ennex.com/fabbers/StL.sht>

[8] International Alliance for Interoperability. 1997. "IFC Specifications Development Guide", IAI, November 1997.

Appendix A: Selections of SCI Code

Below is part of the vectorplot::render method. This part is the section that calculates the arrowheads.

```
void vectorplot::render (datalist *dl, slice ns, int refresh)
{
            ::glBegin (GL_LINES);
            for (i = 0; i != m->ncvx; i++)
                  for (j = 0; j != m->ncvy; j++)
                  {
                        mpointx = m->getcx()[i];
                        mpointy = m->getcy()[j];
                        mpointz = m->getcz()[s.node];
                        vectx = v->getx()->data[XYZ(i, j, s.node)];
                        vecty = v->gety()->data[XYZ(i, j, s.node)];
                        vectz = v->getz()->data[XYZ(i, j, s.node)];
                        double angle1 = atan2 (vectx, vectz);
                        double angle2 = atan2 (vecty, vectx * sin
(angle1) + vectz * cos (angle1));
                        //first do a rotation around angle one with the
                        // y axis, then do a rotation around the x
axis.
                        // this should make the vector have x= 0, y =
0, z = ?

                        double vectdist = sqrt (vectx * vectx +
                              vecty * vecty + vectz * vectz) * sf *
0.125;
                        coldat = 1.0f - (((vectdist / sf / 0.125) - v-
>datamin)/
                              (v->datamax - v->datamin));
                        if ((coldat > 1.0) || (coldat < 0.0))
                        {
                              if (coldat > 1.0) coldat = 0.99999;
                              if (coldat < 0.0) coldat = 0.00001;
                        }
                        rrr = SPECTRAL_PALETTE[int (coldat *
float(SPECTRAL_PALETTE_ENTRIES)) * 3 + 0];
                        ggg = SPECTRAL_PALETTE[int (coldat *
float(SPECTRAL_PALETTE_ENTRIES)) * 3 + 1];
```

```
                        bbb = SPECTRAL_PALETTE[int (coldat *
float(SPECTRAL_PALETTE_ENTRIES)) * 3 + 2];
                        ::glColor3f (rrr, ggg, bbb);

                        ::glVertex3f (mpointx, mpointy, mpointz);
                        ::glVertex3f (mpointx + vectx * sf,
                              mpointy + vecty * sf,
                              mpointz + vectz * sf);
//                      ::glColor3f (0.3, 0.3, 0.3);

                        double ax, ay, az;
                        double tx, ty, tz;
                        ax = 0.8660254 * vectdist; ay = 0.5 * vectdist;
az = 0.0;
                        tx = ax;
                        ty = ay * cos (-angle2) - az * sin (-angle2);
                        tz = ay * sin (-angle2) + az * cos (-angle2);
                        tx = tx * cos (angle1) - tz * sin (-angle1);
                        ty = ty;
                        tz = tx * sin (-angle1) + tz * cos (-angle1);
                        ::glVertex3f (
                              mpointx + (vectx * sf * .75) + tx,
                              mpointy + (vecty * sf * .75) + ty,
                              mpointz + (vectz * sf * .75) + tz
                              );
                        ::glVertex3f (mpointx + vectx * sf,
                              mpointy + vecty * sf,
                              mpointz + vectz * sf);
                        ax = -0.8660254 * vectdist; ay = 0.5 *
vectdist; az = 0.0;
                        tx = ax;
                        ty = ay * cos (-angle2) - az * sin (-angle2);
                        tz = ay * sin (-angle2) + az * cos (-angle2);
                        tx = tx * cos (angle1) - tz * sin (-angle1);
                        ty = ty;
                        tz = tx * sin (-angle1) + tz * cos (-angle1);
                        ::glVertex3f (
                              mpointx + (vectx * sf * .75) + tx,
```

```
                        mpointy + (vecty * sf * .75) + ty,
                        mpointz + (vectz * sf * .75) + tz
                        );
                    ::glVertex3f (mpointx + vectx * sf,
                        mpointy + vecty * sf,
                        mpointz + vectz * sf);
                    ax = 0.0; ay = -1.0 * vectdist; az = 0.0;
                    tx = ax;
                    ty = ay * cos (-angle2) - az * sin (-angle2);
                    tz = ay * sin (-angle2) + az * cos (-angle2);
                    tx = tx * cos (angle1) - tz * sin (-angle1);
                    ty = ty;
                    tz = tx * sin (-angle1) + tz * cos (-angle1);
                    ::glVertex3f (
                        mpointx + (vectx * sf * .75) + tx,
                        mpointy + (vecty * sf * .75) + ty,
                        mpointz + (vectz * sf * .75) + tz
                        );
                    ::glVertex3f (mpointx + vectx * sf,
                        mpointy + vecty * sf,
                        mpointz + vectz * sf);

                }
            ::glEnd ();
            break;
```

The contourplot renderer calls this ::tri for each triangle in the plane. The first three arguments are the first point's xyz location, the next three are the second point's xyz location, the following three are the third point's xyz location, and the last three arguments are the scalar data values for points one, two, and three.

```
void contourplot::tri (double c1x, double c1y, double c1z,
                       double c2x, double c2y, double c2z,
                       double c3x, double c3y, double c3z,
                       double dv1, double dv2, double dv3,
                       double mini, double maxi)
{
    if ((dv2 >= dv3) && (dv2 >= dv1) && (dv1 >= dv3))
        trisort (c3x, c3y, c3z, c1x, c1y, c1z, c2x, c2y, c2z, dv3,
dv1, dv2, mini, maxi);
```

```
      else if ((dv1 >= dv2) && (dv3 >= dv1) && (dv3 >= dv2))
            trisort (c2x, c2y, c2z, c1x, c1y, c1z, c3x, c3y, c3z, dv2,
dv1, dv3, mini, maxi);
      else if ((dv2 >= dv3) && (dv1 >= dv3) && (dv1 >= dv2))
            trisort (c3x, c3y, c3z, c2x, c2y, c2z, c1x, c1y, c1z, dv3,
dv2, dv1, mini, maxi);
      else if ((dv2 >= dv1) && (dv3 >= dv2) && (dv3 >= dv1))
            trisort (c1x, c1y, c1z, c2x, c2y, c2z, c3x, c3y, c3z, dv1,
dv2, dv3, mini, maxi);
      else if ((dv3 >= dv2) && (dv1 >= dv2) && (dv1 >= dv3))
            trisort (c2x, c2y, c2z, c3x, c3y, c3z, c1x, c1y, c1z, dv2,
dv3, dv1, mini, maxi);
      else if ((dv3 >= dv1) && (dv2 >= dv3) && (dv2 >= dv1))
            trisort (c1x, c1y, c1z, c3x, c3y, c3z, c2x, c2y, c2z, dv1,
dv3, dv2, mini, maxi);
}

void contourplot::trisort (double c1x, double c1y, double c1z,
                           double c2x, double c2y, double c2z,
                           double c3x, double c3y, double c3z,
                           double dv1, double dv2, double dv3,
                           double mini, double maxi)
{
      ASSERT ((dv1 <= dv2) && (dv2 <= dv3));

      int l1, l2, l3;
      if (dv1 <= mini) l1 = 0;
       else if (dv1 >= maxi) l1 = this->levels + 2;
        else l1 = floor((dv1 - mini) * (this->levels + 1) / (maxi -
mini) ) + 1;

      if (dv2 <= mini) l2 = 0;
       else if (dv2 >= maxi) l2 = this->levels + 2;
        else l2 = floor((dv2 - mini) * (this->levels + 1) / (maxi -
mini) ) + 1;

      if (dv3 <= mini) l3 = 0;
       else if (dv3 >= maxi) l3 = this->levels + 2;
        else l3 = floor((dv3 - mini) * (this->levels + 1) / (maxi -
mini) ) + 1;
```

```
	::glColor3f (0.0, 0.0, 0.0);

	if (l1 == l2 == l3) return; // Bug out of no contour box.

	for (int i = l1; i != l3; i++)
	{
		double value = double(i) * (maxi - mini) / (double(this-
>levels + 1)) + mini;
		double x1, y1, z1;
		double x2, y2, z2;
		this->linear (c1x, c1y, c1z, c3x, c3y, c3z, dv1, dv3,
value,
			&x1, &y1, &z1);
		if (i < l2)
			// Expand to m-n wall
			this->linear (c1x, c1y, c1z, c2x, c2y, c2z, dv1, dv2,
value,
				&x2, &y2, &z2);
		else
			// Expand to m-x wall
			this->linear (c3x, c3y, c3z, c2x, c2y, c2z, dv3, dv2,
value,
				&x2, &y2, &z2);
		::glBegin (GL_LINES);

		levelpoints[i].push_back(xyznode(x1, y1, z1));
		levelpoints[i].push_back(xyznode(x2, y2, z2));

		::glVertex3f (x1, y1, z1);
		::glVertex3f (x2, y2, z2);
		::glEnd ();
	}

}
```

```
void contourplot::linear (double c1x, double c1y, double c1z, double
c2x, double c2y,
                                        double c2z, double dv1, double
dv2, double level,
                                        double* x, double* y, double*z)
{

      if (c1x == c2x) *x = c1x;
      else
            *x = (level - dv2 + ((dv2 - dv1) / (c2x - c1x) * c2x)) /
                  ((dv2 - dv1) / (c2x - c1x));
      if (c1y == c2y) *y = c1y;
      else
            *y = (level - dv2 + ((dv2 - dv1) / (c2y - c1y) * c2y)) /
                  ((dv2 - dv1) / (c2y - c1y));
      if (c1z == c2z) *z = c1z;
      else
            *z = (level - dv2 + ((dv2 - dv1) / (c2z - c1z) * c2z)) /
                  ((dv2 - dv1) / (c2z - c1z));
}
```

This is the serial routine for the datalist object. Notice how the objects vl, sl, and ml are member objects who the datalist has custody over, hence their serial methods are called.

```
void datalist::serial (CArchive &ar, float ver)
{
      vl.serial (ar, ver);
      sl.serial (ar, ver);
      ml.serial (ar, ver);
      if (ar.IsStoring ())
      {
            ar << buoy << pf << tm << turb;
            ar << steady << beta << cp << density;
            ar << nu << gx << gy << gz;
            ar << trefmin << trefmax;
            ar << unsteady_first << unsteady_output;
            ar << unsteady_delta << unsteady_total;
            ar << unsteady_start << modifyout;
            ar << spec << restart << imon;
```

```
            ar << ipref << jmon << jpref << kmon;
            ar << kpref << maxi << maxr << usemidmon << usemidpref;
      } else
      {
            ar >> buoy >> pf >> tm >> turb;
            ar >> steady >> beta >> cp >> density;
            ar >> nu >> gx >> gy >> gz;
            ar >> trefmin >> trefmax;
            ar >> unsteady_first >> unsteady_output;
            ar >> unsteady_delta >> unsteady_total;
            ar >> unsteady_start >> modifyout;
            ar >> spec >> restart >> imon;
            ar >> ipref >> jmon >> jpref >> kmon;
            ar >> kpref >> maxi >> maxr >> usemidmon >> usemidpref;
      }
}
```

Appendix B: IFC Format Specification

This method description of one IFC server is taken from the users manual for BSPro COM-Server for IFC-Files by the International Alliance for Interoperability.

Objects / Methods supported in the Alfa Version of BSPro COM-Server

These are the classes supported in the Alfa version of BSPro COM-Server, in alphabetic order:

Name of COM Object	Definition from IFC
IfcAxis2Placement3d	The location and orientation in three dimensional space of three mutually perpendicular axes. An *IfcAxis2Placement3D* is defined in terms of a point (inherited from IfcPlacement supertype) and two (ideally orthogonal) axes. It can be used to locate and originate an object in three dimensional space and to define a Placement Coordinate System. The class includes a point which forms the origin of the Placement Coordinate System. Two direction vectors are required to complete the definition of the Placement Coordinate System. The Axis is the placement Z axis direction and the *RefDirection* is an approximation to the placement X axis direction.
IfcBuilding	The Building represents a structure that provides shelter for its Occupants or contents and stands in one place
IfcBuildingStorey	The Building Storey has an elevation and represents a (nearly) Horizontal aggregation of Spaces that are vertically bound. The Building Section is also used to provide a basic structuring Hierarchy for the components of a building construction project (together with side, building, and space), and for the spaces within a building construction project (together with side).
IfcCollection	IFC-entity: none. Collection object
IfcColumn	A vertical structural member which often is aligned with a structural grid intersection. IfcColumn is defined in the Architecture Domain and possibly reused by other domains. It represents a vertical, or nearly vertical structural member designed to transfer loads to its base.
IfcDoor	IfcDoor is defined in the Architecture Domain and possibly reused by other domains. It represents a construction for closing an opening, intended primarily for access.

IfcFace	*Definition from ISO/CD 10303-42:1992:* A face is a topological entity of dimensionality 2 corresponding to the intuitive notion of a piece of surface bounded by loops. ... A face shall have at least one bound, and the loops shall not intersect...
IfcMaterial	Name shortered from IFC-entity "IfcMaterialLayer". Semantic Definition: Substance that can be used to form elements.
IfcMaterialSet	Name shortered from IFC-entity "IfcMaterialLayerSet". A designation by which an element which is constructed from a number of material layers is known and through which the relative positioning of individual layers can be expressed.
IfcObject	The generalization of any semantically treated things and processes within IFC. Examples for IfcObject include physically tangible items, such as wall, beam or covering, physically existent items, such as spaces, or conceptually items, such as grids or virtual boundaries. It also stands for processes, such as work tasks, controls, documents, etc. Objects are independent pieces of information, that might contain or reference other pieces of information, most notably properties. For properties that are shared among multiple instances of an Object the Type driven Property definitions are used, which are shared by value. For properties which values vary for each instance of an Object, the Occurrence Properties are defined. Property definitions can be enhanced to deal with specific national, application type, or project requirements, therefore there is an placeholder for Extended Properties.
IfcOccurrencePropertySet	The IfcOccurrencePropertySet defines a set of type driven properties (and is therefore defined as part of the IFC standard) that have a single occurrence per instance of an semantic object, i.e. each semantic object instance creates its own instance of the IfcOccurrencePropertySet with particular property values.
IfcOpeningElem	Name shortened from IFC-entity "IfcOpeningElement". Opening Element stands for opening, recess or chase, all reflecting voids. It represents a void within any element, that has physical manifestation. Openings can be handled by all sectors and disciplines in AEC/FM industry, therefore the interoperability for Opening Elements is provided at this high level.
IfcOwnerHistory	The *IfcOwnerHistory* defines all history and identification related information. In order to provide a fast access it is directly attached to all independent objects, relationships and properties. The *IfcOwnerHistory* is used to identify both the owning application and user for the associated subject object. In addition the description, provided by the owning user, can be

	kept, which is however only significant to the user. It also references an audit trail to keep the history of the object by which it is contained.
IfcPoint2d	IFC-entity: none. Two dimensional point object
IfcPoint3d	IFC-entity: none. Three dimensional point object
IfcProject	The undertaking of some engineering activities leading towards a product. It acts as the top container for all objects defining a project. The Project also holds the units used for certain measures Throughout the project and the central registry, currently for only team members and applications. The IfcProject establishes the World Coordinate System, WCS.
IfcPropertySet	The IfcPropertySet is a container class which allows the definition of collections of Properties or other Property Sets. It therefore allows for nested list. These lists may than be attached to objects or relationships at runtime. An IfcPropertySet can include other IfcPropertySet's, and it can reference its parent IfcPropertySet, in which it is included. The inherited (INV) PartOfPropertySet Relationship is used to reference the parent IfcPropertySet.
IfcProxy	The IfcProxy is intended to be a kind of a container for wrapping up non-IFC objects for use within the persistent store. Given that we have only a limited number of constructs formally defined within IFC (and will never be able to define them all), we must provide a mechanism for capturing constructs (primarily geometric) that are not defined by IFC. These constructs may or may not have semantic meaning, depending on whether any shape representations or extended property sets are attached to the IfcProxy. Either way, a receiving system only has to ensure that they are maintained as part of the project model. Such a mechanism allows to round trip data that is part of the project but not necessarily part of the IFC model.
IfcServer	IFC-entity: none. COM Server
IfcSimpleProperty	The IfcSimpleProperty defines an attribute class, for which the *value -- name* pair is given. It shall be used for those simple properties, where the unit is already implied by the specific type of the IfcMeasureValue and the IfcUnitAssignment. For simple properties with measures that refers to more specific units, the IfcPropertyWithUnit shall be used.
IfcSite	A defined area of land, possibly covered with water, on which the project construction is to be completed. A site may be used to erect Building(s) or other AEC products. Site may include definition of the single geographic reference point for this site (global position using Longitude, Latitude and Elevation) for the project. This definition is given for informational purposes only. it is not intended to

	provide an absolute placement in relation to the world. The geometrical placement of the Site, defined by the IfcLocalPlacement, has to be always relative to the Project, if the PlacementRelTo attribute is Specified. A project may span over several connected or disconnected sites. Therefore Site Complex provides for a Collection of Sites included in a project. The Site Complex is handled by an IfcGroup having a Group Purpose of "SiteComplex".
IfcSpace	A Space represents an area or volume bounded actually or theoretically. Spaces are areas or volumes that provide for certain functions within a building.
IfcVector3d	Name changed from IFC-entity "IfcVector". The vector is defined in terms of the direction and magnitude of the vector. The value of the Magnitude attribute defines the magnitude of the vector. Note, the magnitude of the vector can not be reliable calculated from the components of the Orientation attribute. This form of representation was selected to reduce problems with numerical instability. For example a vector of magnitude 2.0 mm and equally inclined to the coordinate axes could be represented with Orientation attribute of (1.0,1.0,1.0).
IfcWall	IfcWall represents a vertical construction that bounds or subdivides Spaces. It is the common concept of a wall that will be later specialized in the various domains. In future releases (when structural systems are added to the IFC Object Model) IfcWall will be, e.g., a candidate for a structural load bearing interface.
IfcWindow	Construction for closing a vertical or near vertical opening in a wall or pitched roof that will admit light and may admit fresh air into the adjacent building space.
Preference	IFC-entity: none. Preference class