Naval Research Laboratory
Washington, DC 20375-5320

NRL/MR/6180--99-8367

Development of a Method for Representing Complex Geometries in a Fire Model

J. B. HOOVER
P. A. TATEM

Navy Technology Center for Safety and Survivability
Chemistry Division

April 12, 1999

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget. Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE April 12, 1999	3. REPORT TYPE AND DATES COVERED Interim Report 1998-1999	
4. TITLE AND SUBTITLE Development of a Method for Representing Complex Geometries in a Fire Model			5. FUNDING NUMBERS 61-6000-09
6. AUTHOR(S) J.B. Hoover and P.A. Tatem			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5320			8. PERFORMING ORGANIZATION REPORT NUMBER NRL/MR/6180--99-8367
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research Arlington, VA 22217-5660			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE A

13. ABSTRACT *(Maximum 200 words)*

The Consolidated Fire Growth and Smoke Transport (CFAST) model has frequently been used to simulate fires in structures. However, the basic design of the model, combined with historical restrictions on the type and amount of geometrical information which can be entered into the model, place limitations on the accuracy of the model's predictions. These limitations become especially restrictive in the Naval environment due to the inherent complexity of shipboard architecture. This report documents the development of a alternative method for representing such complex geometries. Our approach, called the Structured Architecture for the Fire Environment (SAFE) uses object-oriented data structures to provide a hierarchical and extensible representation. A prototype data entry application has been written to demonstrate the capabilities of the method and, as a proof-of-concept, a portion of the test ship ex-USS SHADWELL has been described using both SAFE and the standard CFAST method.

14. SUBJECT TERMS Fire Modeling Shipboard architecture CFAST Complex geometry			15. NUMBER OF PAGES 70
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

DEVELOPMENT OF A METHOD FOR REPRESENTING COMPLEX GEOMETRIES IN A FIRE MODEL

1.0 INTRODUCTION

This report documents the design and development of a prototype software system capable of representing complex geometries for use in an advanced zone fire model. We discuss the geometric limitations of the Consolidated Fire Growth and Smoke Transport (CFAST) model when applied to a real-world Navy problem and illustrate those shortcomings using a case study. An alternative approach to representing complex geometries is proposed and a prototype implementation of the concept is presented.

For the benefit of developers, the application programming interfaces (APIs) for the prototype software are discussed in the Appendices. For every class, there is a brief description of the purpose of each public function, followed by the function syntax and a description of the function parameters and return value, if any. Some functions are overloaded so that there may be more than one syntax. Additional comments are also included where further explanation or discussion is needed.

1.1 Fire Model Background

Fire models are often classified as zone models or field models. In a zone model, the region of interest is divided into a relatively small number of zones (typically one or two for each compartment) and detailed physics and chemistry are not included. For example, momentum transfer between zones is typically omitted or grossly simplified and combustion chemistry is generally limited to a simple fuel:oxygen:product balance. Each zone is considered to be a homogeneous volume having time-varying properties which are described by ordinary differential equations. Zone models can be relatively fast (near real time or better) when run on standard microcomputers but, due to their low spatial resolution and simplified physics and chemistry, they can provide only coarse approximations. They are frequently used during design and post-fire investigation when their approximations are acceptable and the alternative, field modeling, would be prohibitively expensive.

In contrast, field models represent the problem as many thousands (or even millions) of cells having properties that vary with location as well as with time, which requires the use of partial differential equations. As a result of the large number of cells and the complex chemistry and physics in each cell, field models can simulate combustion with great accuracy at the cost of very long computation times, even on supercomputers.

Zone fire models were originally designed to simulate simple fires in single rooms and, over many years, the models have been extended to take into account additional physics and more complex geometry. Since each model developer had different perceptions of the relative importance of various phenomena, the capabilities of the models began to diverge as features were added.

Currently, several well established zone fire models are in use, including Harvard 6 [1], FIRST [2], BRI [3] and CFAST [4]. However, because it is capable of handling multi-compartment scenarios and is still actively under development by the National Institute of Standards and Technology (NIST), CFAST has become the baseline against which other models must be compared, at least in the United States[1]. CFAST has been proposed for use by the US Navy for three distinct purposes:

[1] BRI was developed in Japan and is popular there.

Manuscript approved March 18, 1999.

a. evaluation of the fire-safety characteristics of possible new ship designs;

b. providing real-time predictions of fire growth to improve situational awareness during shipboard fires; and

c. fire simulation for damage control training.

1.2 CFAST Description

In this section, we give a brief overview of CFAST and its capabilities. More details may be found in references [4] and [5].

CFAST is a multi-room, zone fire model which may be used to simulate the behavior of enclosed fires. It is capable of predicting air and surface temperatures, mass flow rates through doors and window, the height of the interface between the hot (upper) and cool (lower) atmosphere layers, concentrations of oxygen and key combustion products and other parameters that, collectively, define the environment. This model has typically been used by fire protection engineers to estimate the performance of proposed building designs and as an aid in post-fire investigations.

By default, CFAST uses two zones (layers) to represent the atmosphere of each room[2]. The zones are separated by a horizontal plane and correspond to the hot upper layer and the cool lower layer which are commonly observed in confined fires. The user can override the two-zone default and force the model to use only one zone for selected rooms. This capability improves the model's accuracy in cases where the room is well mixed (for example, in vertical shafts or in rooms far from the fire).

One of the most important parameters which must be defined is the fire size. CFAST uses a specified, rather than a self-consistent, fire — that is, the user must provide inputs that define how the fire size varies as a function of time. This implies that the user must have some insight into the expected behavior of the fire before the model can be run.

Rooms are defined in terms of height, width, depth and floor elevation (relative to an arbitrary baseline). Some thermal and mechanical properties of the wall, ceiling and floor materials (such as thermal conductivity and density) may be specified. However, because the wall is treated as a single entity that wraps around the entire perimeter of the room, the properties of the wall material are assumed to apply to the entire wall area.

Each room may be horizontally connected to any other room or to the outside via up to four vents, representing windows and doors. These vents are defined by ~~their~~ width, sill height and soffit height (heights are measured relative to the floor of the room). It is also possible to specify a time line for vent openings and closings. Vertical vents, which penetrate floors and ceilings, are defined in terms of their areas and shapes (circular and square vents are allowed). In addition to these pressure-induced vent flows, CFAST can also simulate the effects of a network of mechanical ventilation ducts and fans.

It is possible to specify that one room is directly above another, having a common floor/ceiling, but it is not currently possible to define the horizontal relationships among rooms. That is, CFAST

[2] For this reason, CFAST is often referred to as a two-zone model. However, fire plumes and flows through vents are modeled as independent zones, not part of either layer. Therefore, the default configuration of the fire compartment actually has at least three zones (two layers and a fire plume) and possibly more, depending on the number of vents.

does not "know" which wall (if any) are common to two rooms, therefore, horizontal room-to-room conduction can not be modeled at present. Also, the exact horizontal location of vents can not be specified. Due to the wrap-around nature of the walls, this implies that a horizontal vent could actually be in any one of the walls.

Conceptually, CFAST may be viewed as a series of concentric circles, as shown in Figure 1. The user interacts only with the outermost ("Geometry") layer; the other two layers represent parts of CFAST that are invisible and inaccessible to the user. The innermost circle ("Solver") is a mathematical equation-solving engine. At this level, the problem has been reduced to solving a set of simultaneous equations which have no inherent physical meanings.

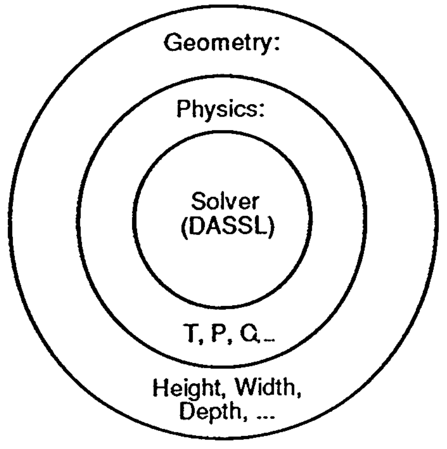


Figure 1. Conceptual View of CFAST

Conceptually, CFAST may be viewed as a set of three concentric circles, with only the outermost accessible to the user. Using the appropriate keywords, the user provides a geometric description of the scenario which is to be modeled. This user-supplied information is translated into a physical description which is then converted into a set of coefficients for mathematical equations which are operated upon by the equation solver.

The intermediate ("Physics") layer provides the physical context, within which the equations do have meaning. That context is based on the user-entered description of the fire scenario, which includes the fire specifications, the room and vent dimensions, mechanical ventilation parameters (if any) and the properties of the construction materials.

1.3 CFAST Limitations

Each of the three layers shown in Figure 1 introduces constraints that affect the range of scenarios that can be modeled, the type of information that can be obtained from the model or the accuracy of the predictions. For example, the "Solver" layer uses the DASSL library, which is limited to solving ordinary differential equations. Thus, even if CFAST could be altered to permit a large number of very small zones, it would still not be able to provide the detailed information that can be obtained from a field model because those calculations require partial differential equations.

Likewise, the equations used to implement the "Physics" layer constrain not only the types of phenomena that CFAST incorporates but also the accuracy to which the effects of a phenomenon can be calculated. As an example, CFAST has always included a correction for atmospheric absorption in its calculation of radiative energy transport. However, until a new absorption submodel [6] was recently added, that absorption calculation did not take into account the concentrations of soot, carbon dioxide or water vapor. By changing the algorithm to include those factors, some new physical effects were added to CFAST and the accuracy of the radiation transport submodel was improved.

Finally, all communications between the user and the model are channeled through the "Geometry" layer. If the desired geometry can not be described in terms that CFAST "understands," then there is no way to input the actual scenario. In this situation, the model can not be directly applied to the target problem and some approximation of the target problem must be used instead. A corollary is that, if the same approximate description can be produced from several different real geometries, then CFAST has no way of knowing what the actual geometry is and the model predictions must necessarily be approximate.

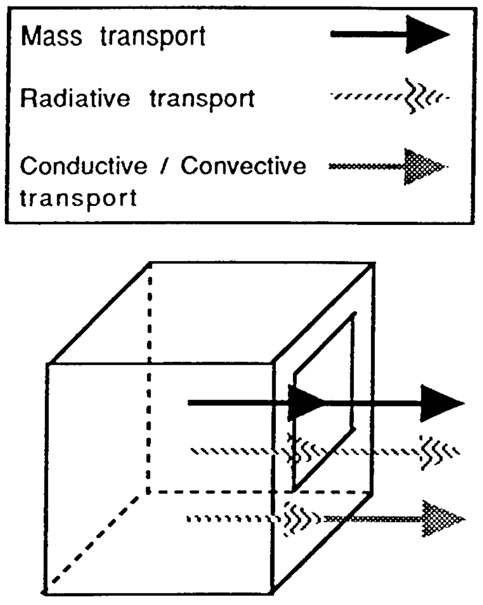


Figure 2. Energy Transport for Simple Single-Room Zone Model

CFAST grew out of a simple, single-room model similar to the one sketched above. Mass transport through vents was supported, as was energy transfer through the boundaries by conduction and convection. Radiative energy transport among the fire, the room boundaries and the atmosphere was also modeled. Energy and mass that left the room, through vents or via conduction, were subtracted from the energy and mass budget of the room.

In order to better understand what limitations exist in current versions of CFAST, and the rationale for their existence, it is useful to briefly review the history of CFAST. The model was originally developed as a single-room model, as illustrated in Figure 2. It was intended to simulate mass transport through horizontal vents (doors and windows), conductive/convective energy transfer at the boundaries and radiative energy transport among the fire, the boundaries (walls, floor and ceiling) and the atmosphere. In order to meet their goal of real-time (or better) modeling, the CFAST developers had to reduce the number of calculations required. This was accomplished by making certain assumptions that had the effects of substituting simple physics for complex physics and approximate geometries for actual geometries.

The required mass and energy transport calculations are computationally difficult for the general case of arbitrary geometry. However, CFAST was primarily intended for use in modeling rooms and, since most rooms are rectangular parallelepipeds, it was reasonable to speed up calculations by using more efficient special-case algorithms. Also, for a single room, energy and mass that leave the room are lost to the surroundings. Since only the interior of the room is of interest in the model, energy and mass fluxes could safely be neglected once they were outside the room. Finally, because the fluxes exiting the room are small compared to what already exists in the ambient environment, the room exterior could be assumed to remain at some constant state. Essentially, fluxes out of the room were subtracted from the room's energy and mass budget and then discarded since they were of no further interest.

When CFAST was later extended to include multiple rooms, the model was modified to do more of the same — in essence, a single room inside a loop. A schematic of a representative two room simulation is shown in Figure 3. For clarity, the shaded boundaries are shown as separate entities but of course, in reality, they are opposite sides of the same wall. The dashed lines indicate that the vent has been defined to connect the two rooms. Vents could also be defined to connect a room to the exterior environment.

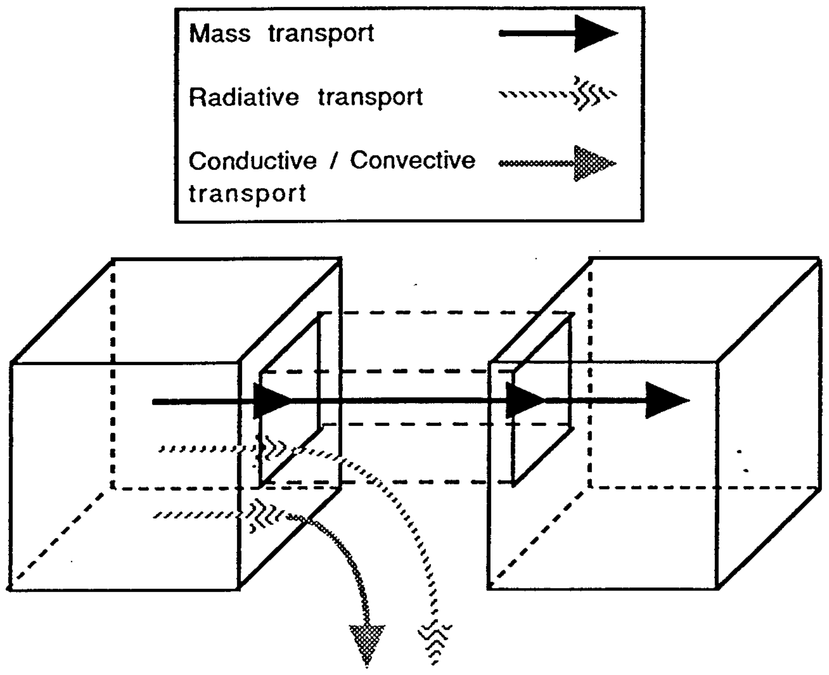


Figure 3. Energy Transport for Simple Two-Room Zone Model

Multi-room capabilities were added to CFAST, but only the mass transport mechanism actually moved energy from one compartment to another. Mass fluxes were subtracted from the source room and added to the destination room. However, energy conducted through the boundaries or radiated through vents was lost to the surroundings.

One of the main purposes of CFAST was to predict the spread of smoke and toxic gases and this clearly could not be done unless room-to-room mass transport was included. Also, due to the thickness and low thermal conductivity of typical building walls and floors, room-to-room conduction is usually negligible. Accordingly, the mass transport submodel was updated to move mass into connected rooms but the radiation and conduction algorithms continued to discard energy leaving the room of origin.

It should also be noted that the rooms are largely independent in the sense that their relative positions are not well defined. Floor elevations and vent connections (such as doors and stairwells) are specified, so CFAST "knows" that the floor of one room is at the same level as the ceiling of another and that there is a doorway between two rooms, but it does not "know" in which wall the door is located[3] or whether room B is to the right, left, front or back of room A. This also implies that CFAST can not "know" that one room is located inside of another — for example, an elevator shaft in the middle of a hotel lobby.

The CFAST developers at NIST have continued to add new features. For example, provisions for mechanical ventilation were added so that multiple compartments could be connected as a network of sources, sinks and fans. Later, when a limited capability for handling vertical, room-to-room conductive heat transport was included, it became possible to specify that one room is directly above another, sharing a ceiling/floor boundary. Thus, CFAST now has two inter-compartment energy transport mechanisms, vents and conduction, but the latter is currently applicable only to the case of a vertical stack (Figure 4)[4]. The requirement for two zones in each room has been relaxed so that rooms may now be represented with a single zone. This often leads to better predictions for rooms which are far from the fire and do not become significantly stratified. Finally, a new smoke transport submodel, known as the hybrid model, was recently added to CFAST to improve predictions for long corridors.

The hybrid model incorporates a correlation function, which was derived from field model simulations of corridor flow, into CFAST. As was alluded to previously, a limited amount of information regarding the relative horizontal location of vents was added to support this feature. However, this information is only available to the hybrid model algorithm, which is only active for the relatively short period during which the smoke front is actually propagating down the passageway. At the end of that period, CFAST reverts to its normal behavior.

To summarize, the goal of fast runtimes forced the developers to simplify (and sometimes neglect) physical phenomena. Those constraints, in turn, imposed limits on the complexity of the geometry that CFAST is capable of modeling. The geometric limitations include the following:

a. all compartments have exactly four walls, one floor and one ceiling;

b. all compartment boundaries are rectangular;

a. no compartment is located inside another compartment; and

b. each wall, ceiling and floor has only one compartment on the "back" side.

---

[3] The recently-added hybrid smoke movement submodel does provide a mechanism for specifying the horizontal offset of a vent relative to reference points in the two connected rooms but this information is only of use to the new algorithm.

[4] Work is now underway at NIST to incorporate horizontal heat conduction from one compartment to another into CFAST.

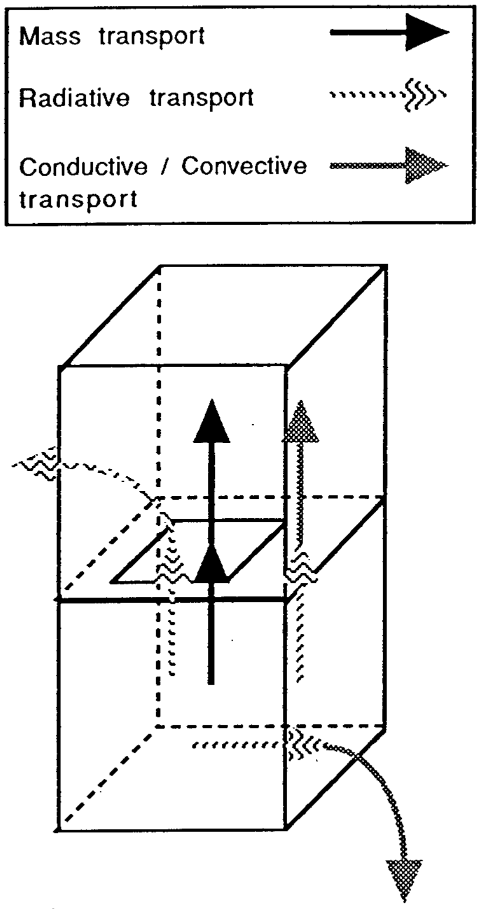


Figure 4. Vertical Energy Transport

Vertical conductivity was added to CFAST so that, in the case shown here, energy can be transported from the lower room to the upper via both mass transport (through the vertical vent) and conduction (through the common boundary). Horizontal conduction and radiative transport through vents still send heat into the surroundings.

The current version of CFAST is much more sophisticated than the original model from which it evolved but, at its core, it still retains many of the original assumptions, including those mentioned above. Those assumptions have proven to be reasonably good for buildings, where rooms typically are simple parallelepipeds, they seldom occur inside of other rooms and where walls, ceilings and floors are sufficiently thick that it usually does not matter what is on the "other" side.

Unfortunately, those assumptions are not always appropriate for situations of interest to the Navy. In particular, the requirement for exactly four walls, each of which must be rectangular, makes it very difficult to adequately model shipboard geometries where odd-shaped compartments are relatively common. The assumption that one room can not be within the bounds of another precludes modeling of spaces-within-spaces (escape trunks running through compartments, for example) and it is not unusual to find bulkheads that have a single compartment on one side and several compartments on the other. As a consequence, CFAST may have limitations, when applied to Navy problems, that do not exist or are insignificant in the civilian community.

As an example, consider the situation represented in Figure 5, which illustrates a portion of the Submarine Ventilation Doctrine Test area aboard ex-USS SHADWELL. As is readily seen, each of the CFAST assumptions discussed above is violated by at least one of the compartments shown in the figure. If we wish to apply CFAST to this problem, we must approximate the geometry by reducing it to terms that CFAST can understand.

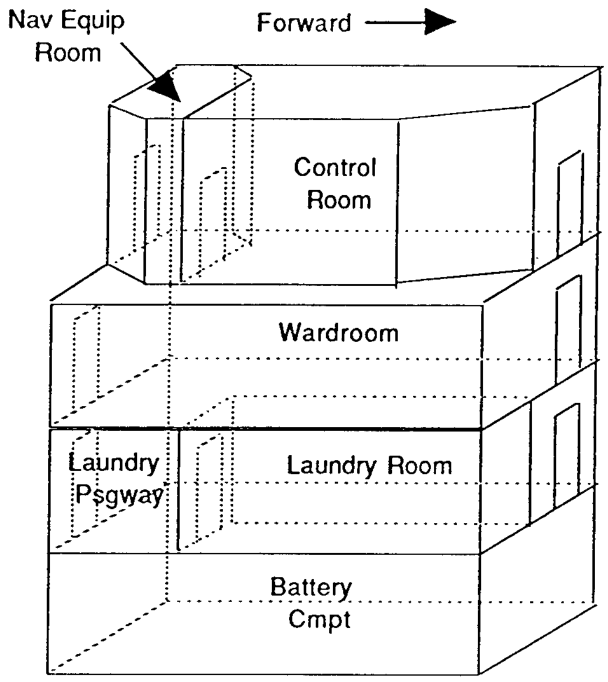


Figure 5. Part of the SHADWELL/688 Configuration

This portion of the SHADWELL/688 test configuration, which was designed to simulate the forward compartment of a USS LOS ANGELES class submarine, illustrates some of the geometric complexity typically found aboard ships. Both the Navigation Equipment Room and the Control Room violate CFAST's constraints on compartment shape; the Laundry is, effectively, located inside another compartment; and there are two compartments directly above the Wardroom.

Essentially, this means replacing each actual compartment with a "similar" compartment that:

a. has exactly four walls, one floor and one ceiling;

b. has only rectangular boundaries;

c. does not contain, and is not contained by, another compartment; and

d. has only one compartment on the "back" side of each boundary.

The exact meaning of "similar" depends on what the user hopes to learn by running the model. Since it is not possible to change the compartment dimensions while simultaneously preserving wall area, floor (and ceiling) area and compartment volume, the user must decide which of these parameters are most important for the specific application.

The process of developing a "similar" compartment description is time-consuming even for experienced CFAST users and always introduces approximations. Depending on the nature of those approximations, the resulting model predictions may be reasonably accurate or very inaccurate.

1.4 Purpose of Current Work

As we have seen, computational limitations have historically driven the choices of which physical phenomena were included in CFAST and what algorithms were used to simulate those phenomena. The algorithm selection dictated the parameters that were required and the user interface, including the geometry model, was designed to provide only those required inputs.

The addition of new physics often created a need for new parameters and the geometry requirements were altered to provide that information. For example, when the heat conduction submodel was changed to include vertical conduction through the ceiling of one room into the room above, it became necessary to add new inputs to specify which compartments were connected.

Rather than indefinitely continue with this *ad hoc* approach, adding new information only as it is needed by the underlying physics, we have investigated the feasibility of an alternative geometry representation that captures all of the available geometric information, whether it is needed immediately or not. Using this approach, modifications to the physics would not normally require changes to the user input. Furthermore, our method is designed to be extensible, so that there is a systematic way of including additional information should that become necessary.

This present work has included preliminary development of a concept, called Structured Architecture for the Fire Environment (SAFE). This concept, when fully developed, would permit direct zone modeling of geometrically complex scenarios (*i.e.*, without making geometric approximations). Because of the strong coupling between the "Physics" and "Geometry" layers in CFAST, a full implementation of SAFE would essentially require a complete rewrite of both layers and is well outside the scope of the current task.

This work should be viewed as a proof-of-concept demonstration of a technology that may become the basis for a future advanced fire model. As part of the current task, we have investigated aspects of the SAFE proposal related to representation of complex geometries in order to determine whether it is actually feasible and we have created prototype software to permit evaluation of the concept.

2.0 CFAST CASE STUDY

The SAFE concept will be discussed in detail in Section 3. However, before doing so, we present a case study that illustrates the limitations of CFAST when modeling real-world Navy problems and which serves as a point of reference for comparison with the SAFE alternative.

2.1 SHADWELL/688 Description

For this example, we have developed a CFAST-compatible description of the SHADWELL Laundry Room and Laundry Passageway areas from Figure 5. Figure 6 shows more details, including compartment dimensions and expanded views of the slanted bulkhead at the bottom of the outboard side of the passageway. The approximation is developed in the following paragraphs and leads to the calculation of effective (*i.e.*, CFAST-compatible) dimensions for the Laundry Passageway.

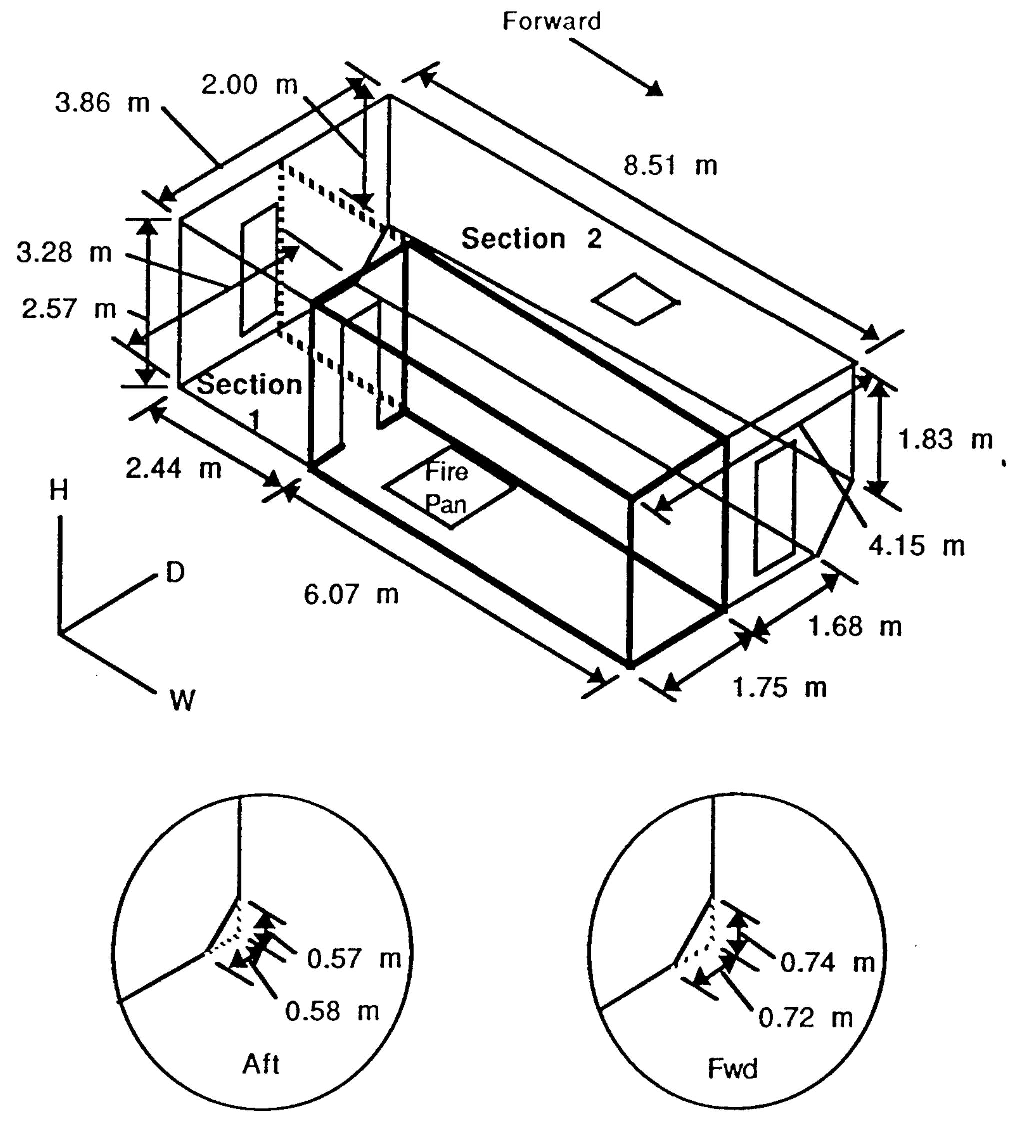


Figure 6. SHADWELL/688 Laundry Room and Laundry Passageway

The layout and dimensions of the Laundry Room (bold outline) and Laundry Passageway used in the Submarine Ventilation Doctrine Test program aboard ex-USS SHADWELL. The dashed lines represent the division of the Laundry Passageway into Section 1 and Section 2, as discussed in the text. Detail drawings at the bottom show the dimensions of the undercut regions at the forward and aft bulkheads.

The Laundry Room itself is a rectangular parallelepiped having dimensions 6.07 m (W) x 1.75 m (D) x 2.57 m (H) and can be exactly represented. The Laundry Passageway must be approximated because:

a. it has seven walls (counting the slanted part of the outboard bulkhead as part of the bulkhead, rather than as part of the deck)[5];

b. the overhead, deck, forward bulkhead and aft bulkhead are non-rectangular; and

c. the Laundry Passageway is partially wrapped around the Laundry Room.

Although it does not pertain to the immediate problem, we should note that the Laundry Room/Laundry Passageway configuration also presents problems for modeling the Wardroom. This is because the Wardroom deck has two compartments (Laundry Room and Laundry Passageway) on the underside, thus violating the constraint that there can be only a single compartment on the "back" side of each boundary.

There are many possible ways that we could approximate the Laundry Passageway but all involve at least three things:

a. remove the fore-to-aft taper;

b. eliminate the slanted region on the outboard bulkhead; and

c. change the floor plane from L-shaped to rectangular.

2.2 Development of the CFAST Geometry Inputs

In order to ensure that our approximation is as faithful to the original geometry as possible, we wish to make the minimum number of changes. We have chosen to conserve volume because that is believed to have the greatest impact on calculations of energy and mass transport from the fire[6]. Therefore, our goal is to calculate effective values for height (H_{eff}), width (W_{eff}) and depth (D_{eff}) such that the resulting effective volume (V_{eff}) is the same as the actual volume (V_{act})

$$V_{eff} = H_{eff} * W_{eff} * D_{eff} = V_{act} \qquad \text{Eqn. 1}$$

Our approach is to first divide the compartment into two parts, Section 1 and Section 2, as illustrated by the dashed lines in Figure 6 so that

$$V_{act} = V_{1,act} + V_{2,act} \qquad \text{Eqn. 2}$$

where V_1 and V_2 are the volumes of Section 1 and Section 2, respectively. Since the heights of the two sections are the same, we use

$$H_{eff} = H_{act} \qquad \text{Eqn. 3}$$

(where H_{act} is the actual compartment height) for both sections.

---

5 This choice was made primarily because both the vertical and the slanted areas were portions of the hull and. therefore, the "back" sides of both were the ambient environment rather than some part of the ship. In CFAST, energy that is conducted through walls disappears from the model, which is the correct behavior in this case.

6 The fire emits quantities of energy and mass in accordance with the user's fire and combustion product specifications. The zone volumes are then used to convert these to energy and mass densities, from which the temperatures and gas compositions may be calculated. Therefore, it is very important that the volumes be correct.

We must find a way to rearrange Sections 1 and 2 so that the floor plane is a rectangle. CFAST has no knowledge of the actual orientation or position of one compartment relative to another, therefore, we are free to choose any configuration that is convenient. For this case study, we have chosen to treat Section 2 as if it were attached to Section 1 at right angles to the Laundry Room, as illustrated in Figure 7. Effectively, we have rotated Section 2 by 90° and reattached it to Section 1, thus changing the Section 2 width dimension to a depth and vice versa.

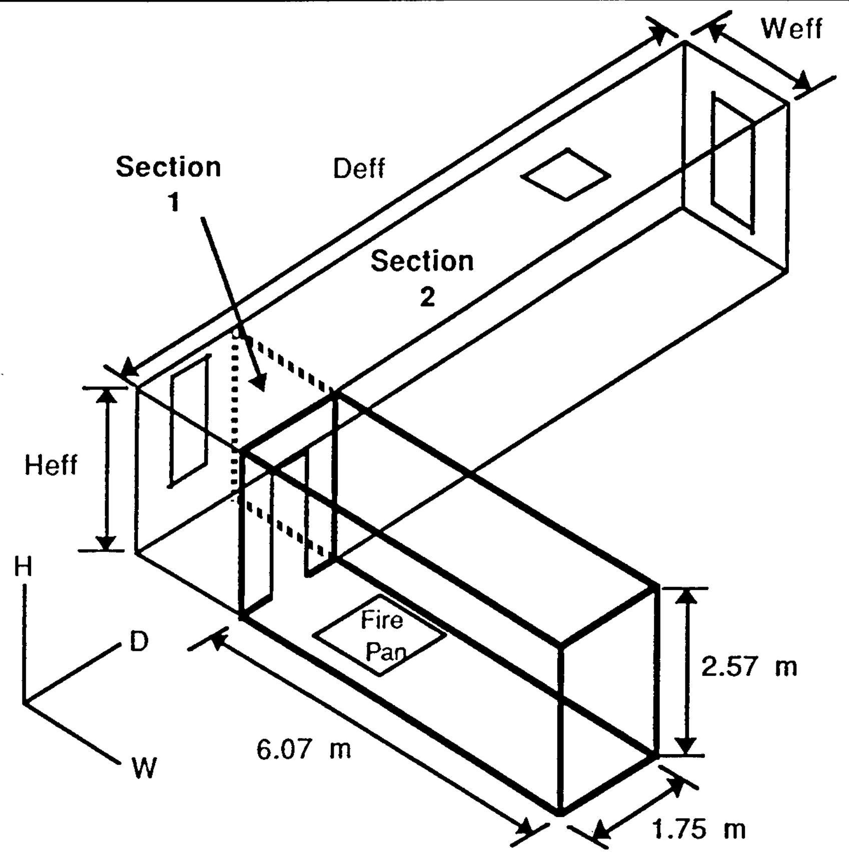


Figure 7. Equivalent Geometry for the Laundry Room and Laundry Passageway

As described in the text, the actual geometry of the Laundry Room/Laundry Passageway can not be exactly represented in CFAST. We produce an equivalent geometry, which can be described in terms understandable to CFAST, by subdividing the passageway into two parts (Sections 1 and 2), using average values for the dimensions of Section 2 and reassembling the parts to make a simpler geometry. The fire compartment is outlined in bold and is identical to that shown in Figures 11A and 11B.

The effective depth is the sum of the depth of Section 1 plus the original width of Section 2

$$D_{eff} = D_{1,act} + W_{2,act} \qquad \text{Eqn. 4}$$

Substituting Equations 2, 3 and 4 into Equation 1, we find the effective width to be

$$W_{eff} = (V_{1,act} + V_{2,act}) / [H_{act} * (D_{1,act} + W_{2,act})] \qquad \text{Eqn. 5}$$

where

$$V_{1,act} = H_{act} * W_{1,act} * D_{1,act} \qquad \text{Eqn. 6}$$

In order to solve this equation, we must calculate the actual volume of Section 2. First, consider the cross-section (in the hd-plane) shown in Figure 8. The coordinates of three points, P_0, P_1, and P_2, are sufficient to define this figure and these coordinates are functions of w, which specifies the location of the cross-section along the out-of-plane axis.

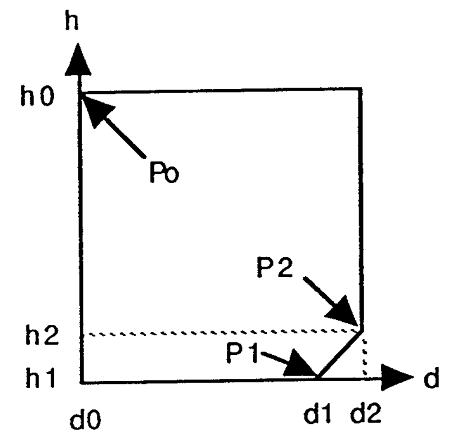


Figure 8. Cross-Section of Laundry Passageway Section 2

The cross-section of Section 2 of the Laundry Passageway (in the hd-plane) may be defined by three points, P_i, having coordinates (d_i, h_i). Note that, in general, these points are functions of the w (perpendicular) coordinate.

From Figure 9, the top and front projections of the Laundry Passageway, we see that the coordinate functions are linear in w, so we define the coordinates as follows:

$$P_0(w) = \{h_0(w) = H, d_0(w) = 0\} \qquad \text{Eqn. 7}$$

$$P_1(w) = \{h_1(w) = 0, d_1(w) = m_{d1} * w + b_{d1}\} \qquad \text{Eqn. 8}$$

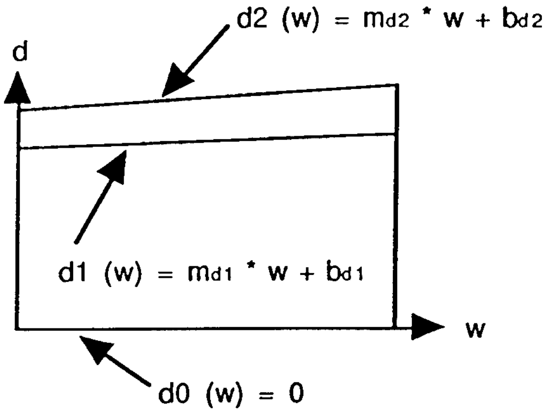
$$P_2(w) = \{h_2(w) = m_{h2} * w + b_{h2}, d_2(w) = m_{d2} * w + b_{d2}\} \qquad \text{Eqn. 9}$$

The area of the cross-section shown in Figure 8 is the area of a rectangles less the area of the cut-out triangle and, as a function of w, is given by

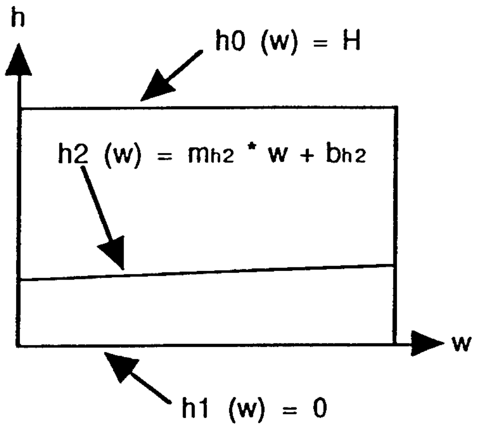
$$A(w) = h_0(w) * d_2(w) - [h_2(w) * (d_2(w) - d_1(w))] / 2 \qquad \text{Eqn. 10}$$

Substituting the functions from Equations 7, 8 and 9, this becomes

$$A(w) = H * d_2(w) - [h_2(w) * (d_2(w) - d_1(w))] / 2 \qquad \text{Eqn. 11}$$



A



B

Figure 9. Top and Front Projections of Section 2 of the Laundry Passageway

As seen in the top view (A), the depths of the defining points for Section 2 are linear functions of w. Likewise, the front view (B) shows that the heights are also linear in w.

The actual volume of Section 2 is then found by integrating this cross-section over the width of Section 2

$$V_{2,act} = \int A(w)\, dw \qquad \text{Eqn. 12}$$

Expanding the expression A(w) and integrating over the range $0 \le w \le W_{2,act}$, we obtain

$$V_{2,act} = [H_{act} * m_{d2} * (W_{2,act})^2] / 2 + H_{act} * b_{d2} * W_{2,act} - [m_{h2} * (m_{d2} - m_{d1}) * (W_{2,act})^3] / 6 - [b_{h2} * (m_{d2} - m_{d1}) * (W_{2,act})^2] / 4 - [m_{h2} * (b_{d2} - b_{d1}) * (W_{2,act})^2] / 4 - [b_{h2} * (b_{d2} - b_{d1}) * W_{2,act}] / 2 \qquad \text{Eqn. 13}$$

where H_{act} is used to emphasize that this value is an actual compartment dimension. The slopes and intercepts in Equation 13 are determined from the coordinates of the endpoints of the lines, using values from Figure 6. By substituting Equations 6 and 13 into Equation 5, we calculate an effective width of a CFAST-compatible approximation of the Laundry Passageway.

Table 1 summarizes the effective dimensions for the Laundry Room, both sections of the Laundry Passageway and the approximated Laundry Passageway. No effective depth was calculated for Section 2 alone because only the value calculated for the entire compartment is meaningful to CFAST. Listing 1 is the geometry portion of a CFAST input file that incorporates this information. The first line of the file specifies the deck elevations relative to a reference. Since this reference level is arbitrary, we have chosen to use the actual deck elevation.

Dimension (m)	Laundry	Psgwy (Sec 1)	Psgwy (Sec 2)	Psgwy (Total)
H_{eff}	2.57	2.57	2.57	2.57
W_{eff}	6.07	2.44	8.51	2.22
D_{eff}	1.75	1.75	---	10.26*

*Note: This value is the sum of D_{eff} for Section 1 and W_{eff} for Section 2, as explained in the text.

Table 1 Effective Dimensions of Laundry Room/Laundry Passageway

Key:
H_{eff} = effective height; W_{eff} = effective width; D_{eff} = effective depth;
--- = not applicable

```
          HI/F            0.00            0.00
          DEPTH           1.75           10.26
          WIDTH           6.07            2.22
          HEIGH           2.57            2.57
```

Listing 1. File for the CFAST-Compatible SHADWELL/688 Example

This portion of a CFAST-compatible input file represents the Laundry Room/Laundry Passageway approximation discussed in the text. The reference elevation was chosen to be the actual deck elevation, therefore the floor height (HI/F) values for both compartments are zero.

2.3 Deficiencies of the CFAST Geometry Inputs

As seen above, a considerable amount of work was required to produce usable inputs. More important than the magnitude of the effort, however, is the fact that the derived values are misleading because some of the geometric parameters that CFAST calculates from those inputs are incorrect and, as a result, the model predictions will be inaccurate. As an example, in Table 2, we have calculated the actual Laundry Passageway deck, overhead and bulkhead areas and compared them with the effective values calculated within CFAST. Discrepancies between actual and effective values are unavoidable when the floor, ceiling or any of the walls are not rectangular.

Effective values were calculated from the dimensions given in Listing 1 where the total bulkhead area is

$$A_{bulk,eff} = 2 * H_{eff} * (W_{eff} + D_{eff}) \qquad \text{Eqn. 14}$$

	Deck (m^2)	Overhead (m^2)	Bulkhead (m^2)
Psgwy (actual)	17.9	23.5	66.2
Psgwy (effective)	22.8	22.8	64.2

Table 2 Comparison of Actual and Effective Laundry Passageway Surface Areas

The effective values are those that are calculated within CFAST, based on the assumptions that the floor, ceiling and all walls are rectangles.

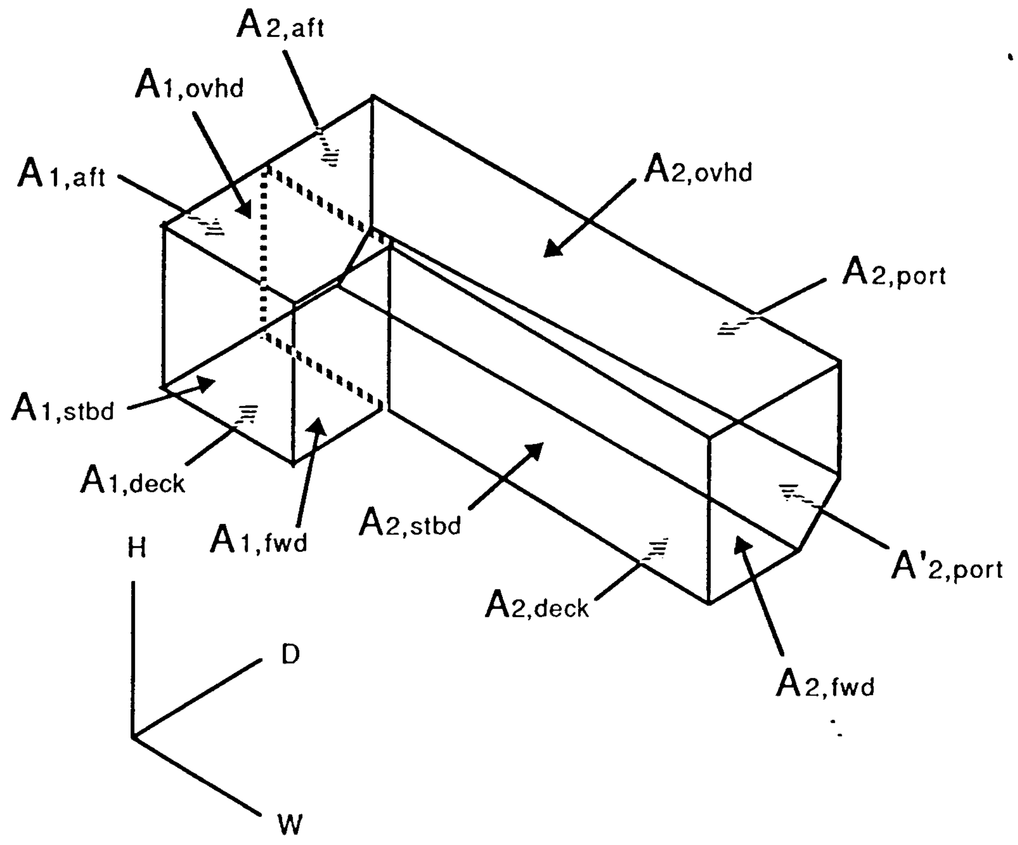


Figure 10. Nomenclature for Calculation of Laundry Passageway Surface Areas

The various pieces of the bounding surface of the Laundry Passageway are labeled for reference. For clarity, the Laundry Room itself has not been shown. Note that there are two parts to the port bulkhead of Section 2.

Actual deck and overhead areas were found by adding the values for Section 1 and Section 2 where the areas for the individual sections were determined from the dimensions shown in Figure 6. Using the terminology illustrated in Figure 10, the actual bulkhead area is

$$A_{bulk,act} = A_{2,fwd} + A_{2,stbd} + A_{1,fwd} + A_{1,stbd} + A_{1,aft} + A_{2,aft} + A_{2,port} + A'_{2,port} \qquad \text{Eqn. 15}$$

The $A_{2,fwd}$ and $A_{2,aft}$ terms were calculated by evaluating Equation 11 for $w = 0$ (aft) and $w = 8.51$ (forward). The other terms were obtained by multiplication of the actual height by the appropriate depth or width dimension from Figure 6.

We noted in the previous section that CFAST has no knowledge of the actual orientations or positions of the two compartments. The only available information regarding relative horizontal positions is that which is implied by the existence of vents (which were left out Listing 1 for simplicity). Clearly, if compartments A and B have a connecting horizontal vent, they must have a bulkhead in common. However, CFAST does not have a method for defining the positions of vents (except for a limited capability associated with the recently-added hybrid smoke movement submodel[7]). Because CFAST treats walls as if they were one continuous, wrap-around entity, rather than four (or more) separate items, it is not even possible to specify in which bulkhead a vent is located.

Due to these limitations, each CFAST configuration represents an infinite number of actual geometries which differ in the positions and orientations of the compartments. As an illustration, Figures 11A and 11B show two alternate scenarios which could be described by the same input file that was created from Figure 7.

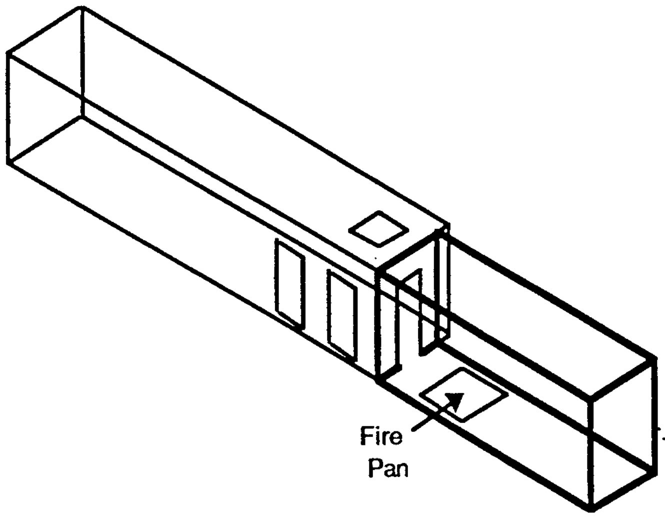


Figure 11A. Alternate Equivalent Geometries

CFAST does not provide a mechanism for defining the actual location of vents, such as doors, or for describing the relative positions and orientations of compartments. Therefore, Listing 1, developed for the situation of Figure 7, could describe this geometry (A). The fire compartment is outlined in bold and is identical to that shown in Figure 7.

[7] This submodel was designed to predict smoke movement in long, narrow corridors and applies only for the time required for smoke to travel to the end of the corridor (typically, a few seconds). At the end of this period, the CFAST reverts to the standard submodel, which does not incorporate information regarding vent positions.

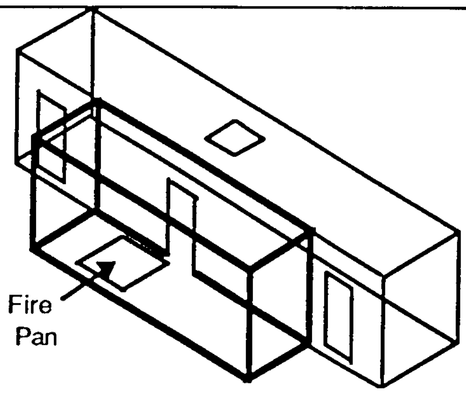


Figure 11B. Alternate Equivalent Geometries (Continued)

Listing 1 also describes this alternate geometry (B). Again, the fire compartment is outlined in bold and is identical to that shown in the previous examples.

This mapping of many possible geometries to one description puts a fundamental limitation on the accuracy of CFAST. Since CFAST can not distinguish among these alternate geometries, and since the physics differs from one case to another, it is impossible for CFAST to apply correct physics in all cases. Instead, it must use some simplified physics which is presumed to be approximately correct for a wide range of situations.

For example, the contact areas between the two compartments are very different in each of the three configurations shown in Figures 7 and 11. As a result, the actual heat transfer rates will be different and we would expect the temperatures to be different. However, since they are all represented by the same model inputs, CFAST will produce the same predictions for all three cases.

3.0 STRUCTURED ARCHITECTURE FOR THE FIRE ENVIRONMENT (SAFE)

A major goal of the SAFE concept was to bypass some of the limitations discussed above by including arbitrary geometry and connectivity in the foundations of a new fire model, rather than attempting to add those features in an *ad hoc* fashion. In this context, arbitrary geometry refers to the ability to directly model compartments of any size and shape (including representation of non-rectangular boundaries, spaces-within-spaces and boundaries which have one compartment on one side and many compartments on the other). Arbitrary connectivity refers to the related ability to track mass and energy transport among the various compartments which may be present in such a complex geometry.

One element which is key to the SAFE concept is the explicit separation of geometry and physics. As was indicated in Figure 1, geometry and physics may conceptually be separated in CFAST but, in the design and implementation of CFAST, they were not actually separated. This commingling of physics and geometry is one reason that it is difficult to add new capabilities to CFAST and is the primary reason that any significant changes in the "Geometry" layer would require a major rewrite of the "Physics" layer in CFAST.

As an example, surface areas are used in many of the physics submodels (*eg*, conduction, convection and radiative heat transfer) and each of the implementing subroutines calculate the relevant area on the assumption that the surface in question is rectangular. Similarly, compartment or zone volumes are frequently used in calculations of species concentrations and other parameters. Again, these calculations tacitly assume rectangular geometry. Changing CFAST to use arbitrary geometries involves not only changing the input functions to "understand" the new descriptions but also finding and changing every occurrence of an area or volume calculation in each of the 300 plus files that comprise the CFAST code.

In addition, there are more subtle interactions between geometry and physics. View factor calculations, for example, involve the relative orientation of emitting and absorbing surfaces and are thus critically dependent on geometry. As a practical matter, geometric assumptions are built into the foundations of CFAST and, as a consequence, removing these dependencies would require the program to be rewritten from the ground up.

The separation of fire physics from geometric elements, as proposed in SAFE, permits modeling domains to be geometrically complex while the physics remains simple enough to be solvable. As more computing power becomes available, the initial physics modules, based on approximate equations, could easily be replaced by more accurate algorithms because the information needed to support those algorithms would already be present.

Another benefit of this separation is that the user interface (which largely represents the scenario geometry) can easily be separated from the actual fire model (which operates at the level of the physics and solver layers of Figure 1). This might allow the user to interact with the model in a more natural way, through graphical depiction of the fire progression instead of through plots of temperature, optical density and similar variables.

3.1 SAFE Geometry Concept Description

A tree-like diagram showing the overall concept for the SAFE geometry model is illustrated in Figure 12. Although technically not a tree (trees can not have multiple arrows pointing to the same object), Figure 12 is sufficiently "tree-like" that we will use standard terminology in which the entities shown as rounded rectangles are called tree nodes (with the topmost one being the root node), arrows point from parent node to child node and nodes which have no children are leaf nodes.

The root node represents the concept of a structure, which encompasses the fire modeling domain — that portion of a building or ship that is of interest to the modeler. It could be the entire entity, but is more likely to be only a portion of it, such as several floors or the section between two specified frames. The structure is assumed to be composed of one or more compartments, each of which "belongs" to that structure (the arrows in Figure 12 may be read as "is defined by" or "contains" — that is, a structure contains, or is defined by, compartments). A compartment is an enclosed space of essentially arbitrary shape. By this definition, corridors and stairwells are considered to be compartments.

To help clarify the concepts, Figure 13 provides a simple example which is used in the discussion below. Figure 13A shows the entire structure which, in Figure 13B, has been resolved into its component compartments, labeled "Cmpt 1" through "Cmpt 3."

Compartments are defined by the bounding planes (partitions) which separate one from another. These boundaries roughly correspond to walls, ceilings and floors but the correspondence is only approximate because several partitions may be needed in cases where the actual boundary is not flat. For example, two partitions (designated as 3-3 and 3-4 in Figure 13C) are needed to represent the right-hand wall of Cmpt 3 whereas the other walls, the ceiling and the floor each require only a single partition.

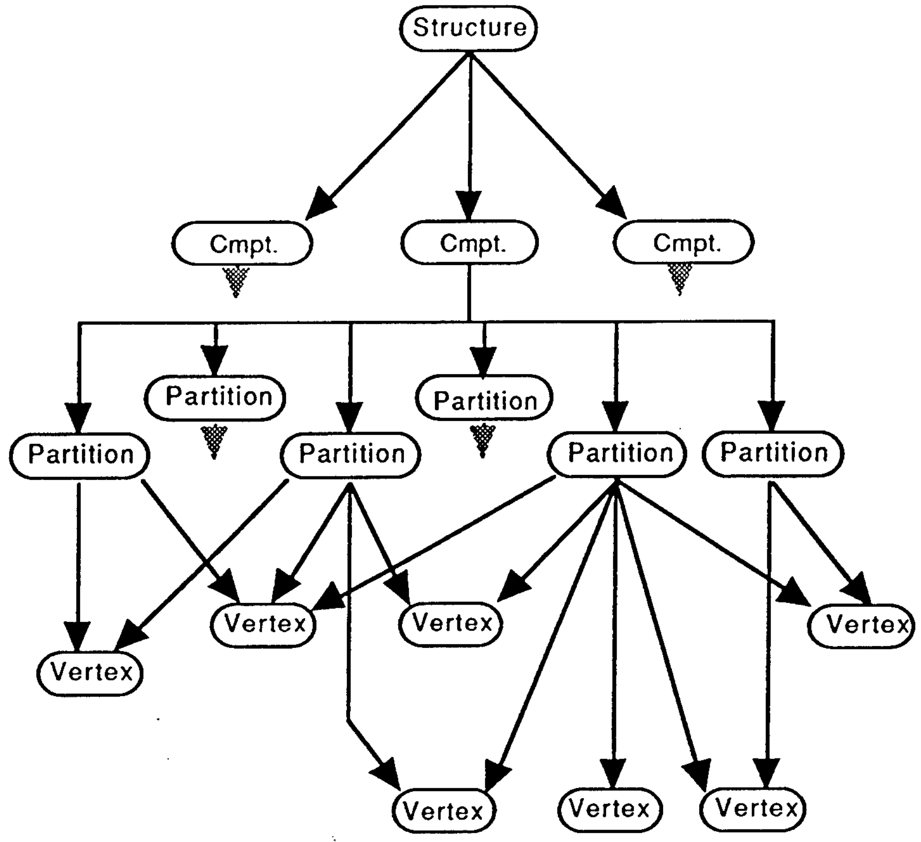


Figure 12. The SAFE Geometry Representation

The SAFE geometry starts with the overall structure which is to be modeled. This is decomposed into compartments, which are individual enclosed spaces. These, in turn, are constructed from partitions, representing walls, ceilings and floors. Each partition is defined by a set of vertex points. Structures, compartments, partitions and vertices are geometrical object types and, except for the vertex points, each of these objects "belongs" to only one higher-level object. Vertices may be shared among two or more partitions.

Also, since a partition "belongs" to one specific compartment, a single boundary is represented by at least two different partitions, one for each side. Figure 13C shows partition 2-5 as the floor of Cmpt 2 and 3-7, as the ceiling of Cmpt 3. In reality, of course, these are opposite sides of the same boundary. It is also possible for one side of a boundary to be composed of a single partition while the other side is represented by several partitions. An example of this situation may be seen in partition 1-3, which has two partitions on the "back" side (2-1 and 3-1).

Each partition is defined in terms of sets of vertex points which specify the three-dimensional coordinates of the corners of the partition. Unlike partitions, each of which is unique to a compartment, a vertex may be shared among any number of partitions. This was done to reduce file size and improve data integrity — only a single copy of each vertex need be kept and a change in the value assigned to it is automatically reflected in all partitions which depend on that vertex.

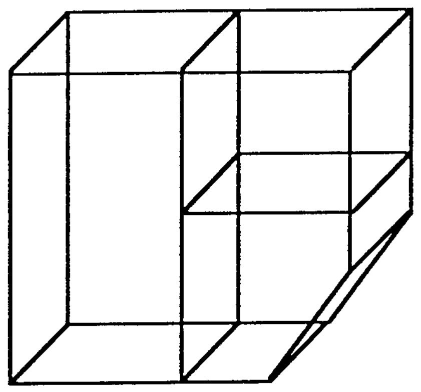


Figure 13A. Examples of Structures, Compartments and Partitions

This structure, a schematic of part of a ship, is the object at the root of the geometry and includes the entire domain which is to be modeled.

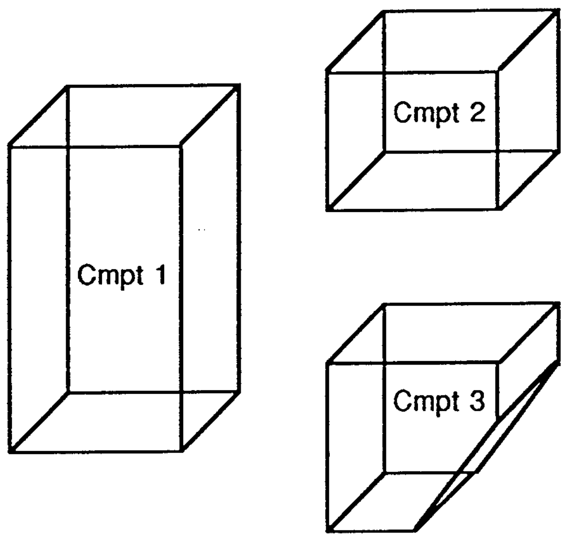


Figure 13B. Examples of Structures, Compartments and Partitions (Continued)

These three compartments "belong" to the structure of Figure 13A. Alternatively, that structure may be considered to be defined by the compartments it contains.

At this point, one might reasonably ask the question: Why develop a new method of representing geometry when there are many computer-aided design (CAD) programs which already do an excellent job of displaying and editing geometry?

The most fundamental answer is that CAD programs were never designed to capture all of the information that is important to fire modeling. A second answer is that those programs, and the geometry description methods that they use, are proprietary. Even in the cases where the format specifications are available, they are subject to change at any time by the owners of the software. As a result, it is likely that significant efforts would have to be invested to ensure continued compatibility with the CAD formats. The format presented here is specifically designed to represent information critical to fire modeling and to be extensible as new capabilities are added to our fire models. It has the further advantage of being exclusively under the control of the Navy.

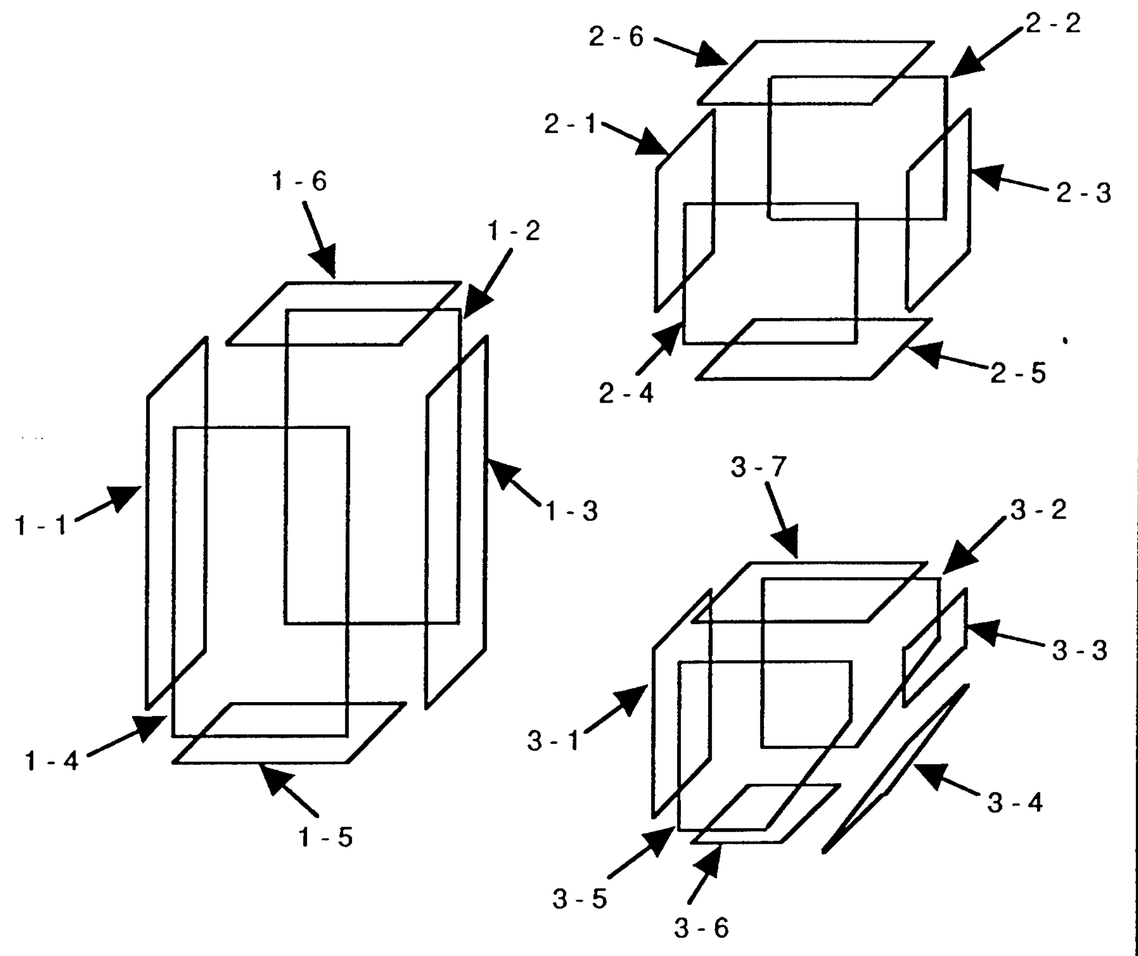


Figure 13C. Examples of Structures, Compartments and Partitions (Continued)

The compartments of Figure 13B may be decomposed into their constituent partitions, as shown in this exploded view. Cmpt 3 has seven partitions, two of which (3-2 and 3-5) are not rectangular.

3.2 Object-Oriented Programming Background

Our approach to software development has been based on object-oriented programming (OOP) practices. In this section, we briefly summarize OOP concepts and advantages. Readers desiring more detailed information are referred to the numerous programming texts, such as references [7] and [8], which discuss these subjects in depth.

The defining characteristics of OOP are generally considered to be encapsulation and inheritance. Encapsulation refers to the ability to define programmatic entities (classes) which contain both data and functions that act on that data. Variables and functions that belong to a class are said to be members of the class.

The most important property of a class is that it is effectively a "black box" whose members are, by default, invisible and inaccessible to the class user. Access to this hidden data is provided only through member functions that have been made publicly available specifically for this purpose by the class developer[8]. This provides a mechanism for error checking that is enforced by the compiler — it does not depend on the class user to do anything. Compared to traditional designs, encapsulation provides better protection against data corruption and inadvertent side effects introduced by changes to the program.

A class may be thought of as a template from which an individual object may be constructed. Essentially, a class describes a type of object — what data it contains, what functions are supported and which parts can be accessed from "outside" the class. An object then represents a specific item of the class. For example, our pCompartment class includes place holders for a compartment name, a list of bounding partitions and related information. A pCompartment object then contains the name of a particular compartment, along with a list of the specific partition objects that describe the boundaries of that compartment and other information specific to that compartment.

Some OOP languages, including C++, have the concept of constructor and destructor functions. The class constructor is automatically called when an object of the class is created, which may occur either statically (when the program was compiled) or dynamically (when the program is executed). The constructor normally is used to initialize the class member variables. Since the constructor call is automatic, the user does not have to do anything to ensure that a new object is properly initialized.

Likewise, the compiler calls destructors whenever an object is destroyed. Destructors are especially valuable when the class contains pointers to data because they can be written to automatically dispose of the pointed-to data. This reduces (but does not eliminate) the possibility of memory leaks that can occur when pointers are thrown away before the memory to which they point has been deallocated.

Constructors and destructors have another useful property — they can be made inaccessible to the user so that any attempt to create or destroy an object results in a compiler error. For example, we have used this technique in the pCompartment class so that compartments can only be created or destroyed at the request of the pStructure object (pStructure is the only class that has been given permission to use the pCompartment constructor and destructor). This ensures that, whenever a new compartment is created, it is properly added to the structure's compartment list and that, when it is deleted, the compartment is removed from the list. As a result, the integrity of the compartment list is preserved.

The second defining characteristic of OOP, inheritance, refers to the ability to define one class (a child class) in terms of another (the parent class) so that the child automatically possesses the data and functions of the parent. This is useful when we want to add new features to a program by extending the capabilities of an existing class. Since the original class is not changed, existing code that uses the parent class will continue to function normally. Developers can then concentrate on ensuring that the new class behaves correctly, instead of on repairing functions that may have been broken by the update.

OOP is especially useful in a project such as this one, where there are clear correspondences between conceptual (program) classes and some actual, physical objects (such as compartments, walls, ceilings and floors). This makes it much easier to decide what classes will be needed in order

[8] In this context, we consider a class developer to be the person(s) who created the class whereas the class user is any person who writes programs that call upon the functions included in the class. The class user should not be confused with the end user of the software, who only runs the program.

to adequately describe a physical system. Knowing how the actual objects behave is also helpful in determining what functions are needed to provide the equivalent behaviors in the software.

3.3 SAFE Geometry Implementation

The geometry model has been written in the C++ programming language, primarily because it is popular and widely supported. Our implementation uses only ANSI standard features, including the Standard Template Library (STL), and should be portable to any compiler which supports ANSI C++.

For this project, we have created the classes pStructure[9], pCompartment, and pPartition to represent structures, compartments and partitions, respectively. Additional classes, u3DVector uUDGraph and uUDGraphNode, were defined to provide utility functions. The latter three classes are generic in that they implement mathematical concepts that are not unique to our immediate requirements. For example, u3DVector can be used to represent vectors whose components are integers, floats, longs or any other numeric type. When the class is used, it is only necessary to specify which underlying type is desired so that the compiler can allocate the proper amount of memory for each u3DVector object.

At present, our implementation of the SAFE geometry model does not use inheritance. However, we have defined two classes (p3DPoint and pVertex) that are place holders for future subclasses. Currently, p3DPoint is defined to be a floating point version of u3DVector and pVertex is defined as a uUDGraphNode having a p3DPoint as its underlying data type. If it should become desirable to add new features to p3DPoint or pVertex, we can simply redefine them to be subclasses instead of synonyms. Each subclass would inherit the properties of its parent class and we could then add whatever additional properties we need. This change would be transparent to the geometry classes — all existing code that now uses p3DPoint or pVertex would continue to work without change.

As an example, u3DVector does not define less than or greater than operators. This is because those operators are undefined for true mathematical vectors — if A = {0, 0, 1} and B = {1, 0, 0}, the vectors are clearly not equal (although their magnitudes are equal) but neither is one larger than the other. However, we can foresee that it may be useful to be able to sort pVertex objects (*eg*., to arrange vertices in an ordered list for easy display).

To accomplish this, we could customize u3DVector by artificially imposing a less than/greater than property, but that would make the class unusable as a general representation of vectors. A better solution is to change p3DPoint to be a subclass of u3DVector and then customize p3DPoint. The parent class, u3DVector, is unaltered and the geometry classes can continue to use the vector properties of p3DPoint without change.

In the sections that follow, we will outline each of the classes that have been developed, starting with the utility and place-holder classes and then working down the geometry hierarchy from structures to partitions. Finally, we discuss a simple application program that was written for test purposes and which permits geometry-related data to be entered.

[9] The "p" prefix is used to distinguish the classes created specifically for this project from generic utility classes (prefix "u") which may be used by many projects. The choice of this naming convention was arbitrary and has no effect on the behaviors of the classes.

3.4 Proof-of-Concept Geometry Demonstration Program

3.4.1 3-D Vectors

Early in the development process, it became clear that it would be necessary to have a generic method for representing points in three-dimensional space. An anticipated requirement was the ability to perform mathematical operations on these points. For example, it was recognized that it would be useful to be able to calculate the distance between two points and to determine whether selected points were coplanar. In short, it was desirable that the vertex points in our geometry model behave as mathematical vectors in three dimensions. Accordingly, the first class to be written was u3DVector, which implements this behavior.

Class u3DVector provides the following capabilities:

a. creation of a vector from three scalar values;

b. copying an existing vector;

c. reading and writing each of the individual coordinates of a vector;

d. calculation of the magnitude of a vector;

e. normalization to a unit vector;

f. vector addition and subtraction;

g. multiplication of scalars and vectors;

h. dot product vector multiplication;

i. cross product vector multiplication;

j. division of a vector by a scalar;

k. vector equality and inequality comparisons; and

l. vector input and output.

3.4.2 3-D Points

The components of the u3DVectors are defined as generic data types. For this project, a single precision, floating point instance of the u3DVector class, p3DPoint, was defined. The latter type has been used throughout the project. The major benefit of using an intermediate data type is that, simply by changing the line of code which defines p3DPoint, our geometry model can be changed to use integers, double precision or some other numerical format.

In addition, we may easily add new behaviors (beyond those of a mathematical vector) to our definition of p3DPoint simply by making it a subclass of u3DVector. Using an intermediate type also permits the basic u3DVector class to be very general so that it can easily be reused in other projects where vectors of other types may be needed. Both of these are examples of the benefits of developing programs with OOP techniques using C++.

3.4.3 Graphs and GraphNodes

The p3DPoint class, described above, gives us the ability to represent points in space. However, this is not sufficient — we must also be able to specify how these points are connected to each other to form the edges which delineate partitions. Figure 14 shows the vertex points which define the Laundry Room and Laundry Room Passageway aboard ex-USS SHADWELL. Compare this with Figure 6 to see the difference between representing a structure as a set of points and as a set of points-plus-connections. Clearly, the former method is virtually useless.

Networks of inter-connected points are known as graphs and have been well studied by mathematicians. For example, graph theory has been applied to problems involving the most efficient routing of traffic in a communications system and the best way to schedule visits on a sales route. Such networks are discussed in terms of graph nodes, which have some specific value (such as a location), and connections between pairs of nodes. Connections may or may not have an associated value (typically, a connection value would be expressed as the cost of using that connection). Note that there can be multiple connections between any pair of nodes, just as there may be multiple routes between pairs of cities.

Cyclic graphs are those in which it is possible to travel from node A to node B and return to node A without retracing your path. In acyclic graphs, such round trips are not possible. Directed graphs have one-way connections while undirected graphs have bi-directional connections.

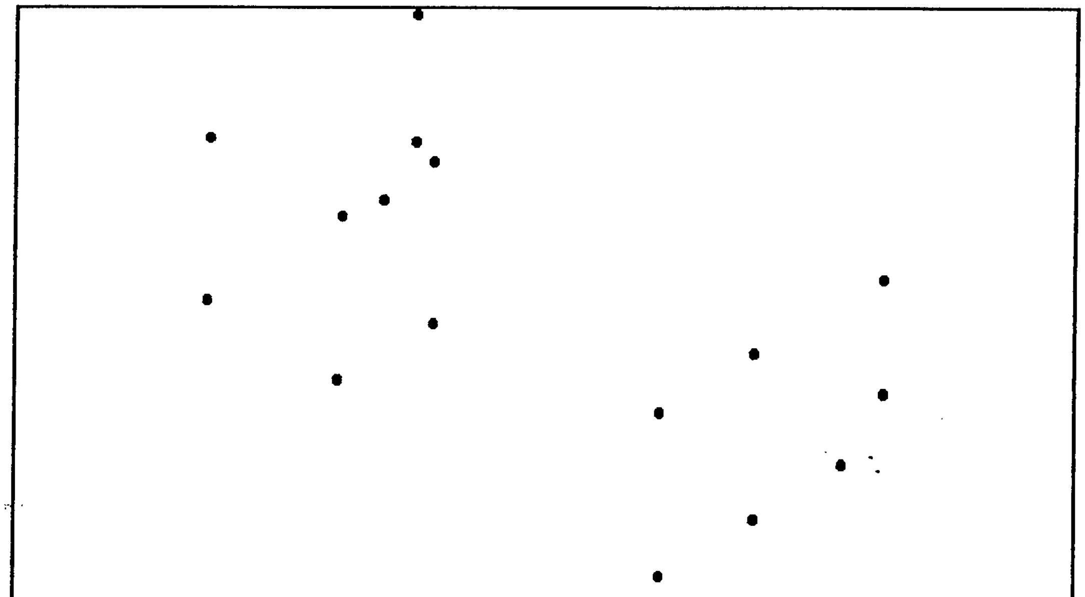


Figure 14. A Compartment Viewed as Points

The set of points shown in this figure is a representation of the Laundry Room and Laundry Room Passageway used for the Submarine Ventilation Doctrine test program aboard ex-USS SHADWELL. This is the same area as was shown in Figure 6.

If we consider the vertices shown in Figure 14 to be the nodes of a graph, there clearly are multiple paths between pairs of nodes and there is no obvious reason to believe that traversals are limited to one direction. In fact, if we describe the compartment boundaries (partitions) in terms of ordered sets of vertex points we quickly find that it is impossible to develop a complete, self-consistent description using directed graphs. At some point, in order to close a loop describing one partition, we must traverse a previously-used connection in the reverse direction, as is shown in Figure 15. Accordingly, for our purposes, we must use an undirected, cyclic graph.

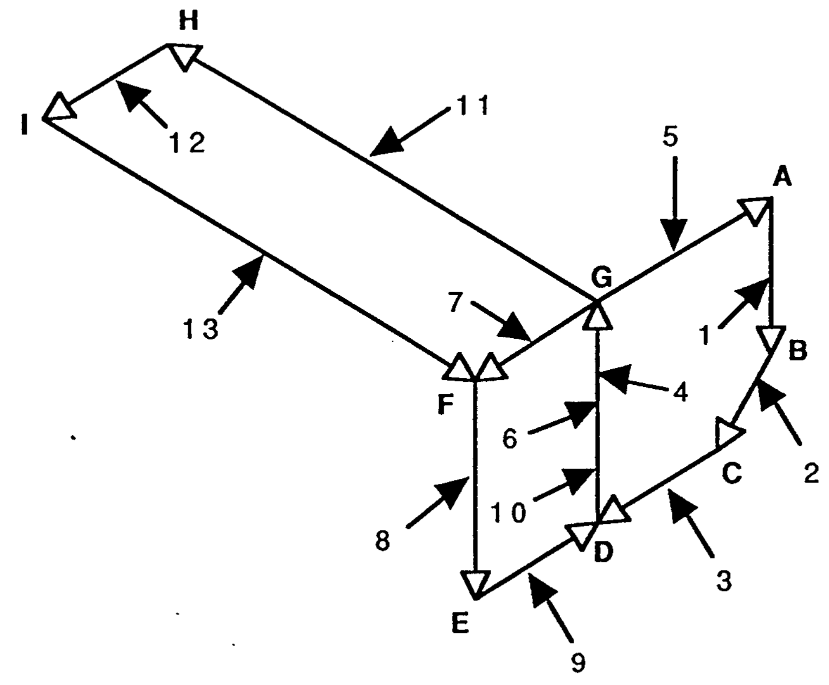


Figure 15. Directed Graph Representation of the SHADWELL/688 Laundry Room and Laundry Passageway

Any attempt to represent the Laundry Room and Laundry Passageway using directed graphs will eventually result in an inconsistency. In this example, vertex points are identified by letters and connections are numbered in the order in which they were created. In order to complete the cycle {G, H, I, F}, connection 14 must be F → G. However, this conflicts with the previously established direction G → F (connection 7).

We have implemented these graphs using two interrelated classes. One class, uUDGraph, represents the entire graph; the other class, uUDGraphNode, represents individual nodes in the graph.

The uUDGraphNode class contains the node value, a node serial number, a list[10] of pointers to those uUDGraphNode objects which are connected to the current node and provides functions for:

a. reading the node serial number;

b. reading and writing the value of the node;

c. iterating through the list of connections associated with the node;

[10] In this project, lists have been implemented using the STL list template, specialized for whatever specific data type is desired. In the case at hand, uUDGraph contains a list of pointers to uUDGraphNode objects.

d. counting the number of connections and testing for empty connection lists;

e. graph node equality and inequality comparisons; and

f. graph node input and output.

There are several features of the uUDGraphNode that require an explanation. First, note that the above list contains no methods for creating new nodes, setting connections between nodes or disconnecting nodes. This is because access to those functions has been restricted so that they are only available to uUDGraph objects. In other words, if you want to create a graph node you must first create a graph object and call one of its methods to add a node. Likewise, changes to the connections among existing graph nodes are accomplished by sending commands to the graph object, rather than to the node objects. This is an example of the use of encapsulation, which is discussed in Appendix A.

These access restrictions were designed into the implementation of graphs in order to ensure data consistency. If they did not exist, it would be possible for a user to create a graph node that is not part of a graph. Such an isolated point makes no sense, because graph nodes have no meaning outside the context of a graph, and could lead to data corruption.

Second, serial numbers were added to the graph node data structure to support input and output. Connections are maintained in the form of pointers to objects and, since the values of these pointers are assigned dynamically during program execution, they can be (and generally are) different each time the program is run. Thus, any pointers that may have been saved from one run will be invalid if the program is run again and the graph is read back from a file. The use of serial numbers permits each node to be unambiguously identified so that the connection network may be correctly reestablished each time the file is read. The serial number must be unique to ensure that it identifies one and only one object. To avoid the possibility that the user might inadvertently change it, it is a read-only data member.

Class uUDGraph contains a list of the graph nodes, also represented as pointers to uUDGraphNode objects. The following capabilities are provided by class functions and operators:

a. construction of an empty graph object;

b. copying an existing graph;

c. adding a new node, with or without a connection to an existing node;

d. deleting an existing node;

e. finding an existing node by its value or serial number;

f. determining whether a node pointer points to a member of the current graph;

g. adding new connections between two existing nodes;

h. deleting connections between two nodes;

i. iterating through the list of nodes;

j. counting the number of nodes in the list and testing for empty node lists;

k. graph equality and inequality comparisons; and

l. graph input and output.

It should be noted here that the node addition function does not permit multiple nodes having the same node value. When a node with a unique value is added, a pointer to the new uUDGraphNode object is returned and may be used in subsequent operations on that object. If, however, an attempt is made to add a node having the same value as an existing node, nothing is added to the node list and a pointer to the original node is returned.

The connection addition function does permit multiple connections between the same pair of nodes. The rationale for this difference in behavior is that nodes represent unique entities, whereas connections are not necessarily unique. For example, if uUDGraph were used in a program to represent a traveling salesman problem, there would be only one node for each city, but there most likely would by many roads connecting each city pair.

3.4.4 Vertices

Vertices have been implemented as a pVertex data type which is a uUDGraphNode specialized to contain values of type p3DPoint. This is analogous to the manner in which p3DPoint itself was defined as a floating point version of u3DVector. Again, the primary reason for using an intermediate data type is to isolate the behavior of the specialized type (pVertex) from the generic behavior of the underlying data type (uUDGraphNode). As discussed above, this permits us to make changes in pVertex without altering uUDGraphNode.

3.4.5 Structures

The pStructure class contains the name of the structure, a list of the compartments which belong to the structure and a graph representing all of the vertex points associated with the structure. The vertex graph may be considered to be a master list of all vertex points (and interconnections) to which other objects may refer. The alternative approach, in which each object maintains a private copy of the data, is prone to data corruption because the copies can become desynchronized — that is, different copies can have different values. Also, the difficulty in making changes in multiple copies more than offsets any inconvenience due to having to work through data pointers instead of directly with the data.

This class includes methods for:

a. creation of an empty structure;

b. copying of an existing structure;

c. reading and writing the structure name;

d. finding an existing vertex by its value or serial number;

e. iterating through the vertex points;

f. counting the number of vertex points in the graph and testing for an empty graph;

g. creating, adding and deleting compartments;

h. finding a compartment by its serial number;

i. iterating through the list of compartments;

j. counting the number of compartments and testing for an empty compartment list; and

k. structure input and output.

3.4.6 Compartments

The pCompartment class includes the compartment name, serial number and a list of the partitions which bound the compartment. The purpose of the serial number is to ensure an unambiguous method of referring to a specific compartment that remains valid from one invocation of the program to the next.

Functions and operators for the pCompartment class include:

a. reading the compartment serial number;

b. reading and writing the name of the compartment;

c. creating, adding and deleting partitions;

d. finding an existing partition by serial number;

e. iterating through the list of connections;

f. counting the number of connections and testing for empty connection lists;

g. compartment input and output.

The compartment serial number is read-only, for the reasons discussed in connection with graph nodes. Note that the creation, addition and deletion of compartments is done through the interface to pStructure, not through pCompartment. This is similar to the approach discussed previously, in which operations on graph nodes are performed by sending requests to the graph object. The reason is the same: internal consistency of the data is maintained by disallowing manipulation of compartments outside the context of a structure. Similarly, pCompartment provides methods for operating on the partition objects contained within the compartment.

3.4.7 Partitions

As might be expected, the pPartition class data includes fields for a partition name (for example, this could be used for a frame number if the partition represents part of a shipboard bulkhead), a serial number and an ordered list of the defining vertex points. Methods are available for performing the following operations:

a. reading the partition serial number;

b. reading and writing the partition name;

c. creating, adding and deleting vertex points;

d. finding an existing vertex by value or by serial number;

e. connecting and disconnecting pairs of existing vertex points;

f. iterating through the list of vertex points;

g. counting the number of vertex points and testing for empty vertex lists; and

h partition input and output.

There is a subtle distinction between the deletion operations in item (c) above and similar,functions in the classes previously discussed. For all other classes, the add and delete functions are inverses: the former causes one object to be added to a list and the latter causes one object (identified by a pointer to the object) to be deleted. This is feasible because the objects are not required to be in any particular order, so arbitrary deletions do not affect the validity of the list.

In the case of pPartition, however, the vertex points are ordered — they represent the sequence in which the vertices would be encountered if the perimeter of the partition were continuously traversed in one direction. In this case, deleting a vertex could invalidate the vertex list. To ensure that this does not happen, the deletion function deletes all vertices in the list and forces recreation of the entire list. The addition function, as usual, appends a single vertex to the list. Note that the use of an append operation, rather than an insert, implies that vertices must be added in the correct sequence.

3.4.8 Data Entry Program

As was mentioned previously, a prototype data entry program was written to test the geometry model. This program was originally used to verify the functions of the various classes as they were being developed and, ultimately, to exercise the entire set of classes. It has a very primitive user interface based on a Unix-like command line menu system and it is anticipated that, for an actual fire model, it will be completely replaced by a modern, graphical user interface. However, the prototype is adequate for the immediate goals of testing the software and validating the geometry model concept. The features of this program are illustrated in Listings 2 - 5, discussed below, which show the available menu options.

It has been noted that the interface for each of the geometry classes, pStructure, pCompartment and pPartition, provides a means of creating and destroying objects of the next lower type in the hierarchy. For example, a function in pStructure is used to create pCompartment objects and a function within a pCompartment object is used to create pPartitions.

Making use of this behavior, the demonstration program allows the user to create and edit a pStructure, including adding and deleting compartments. Once a compartment has been added, another level of the program becomes accessible, permitting editing of pCompartment objects. This editor allows partitions to be added and, again, a lower level editor is then available to edit the pPartitions. At this level, vertex points may be added to specify the perimeter of the partition.

A close inspection of the functions of each of the geometry classes reveals that the pStructure class has no method for creating vertex points even though the master vertex list is a part of the class. This was done for a purpose: it was found that attempting to add vertex points in one place (the pStructure) and establish the connections among them in another (the pPartition) led to confusion and error. Therefore, the creation of vertex points was deferred to the pPartition class. At that level, the user is free to concentrate on a small set of points - those that mark the perimeter of one, simple bounding surface.

The vertex addition function in pPartition first adds the point to the master list, then adds the returned pointer to the local vertex list. Because there can be only one node with a given value, the returned pointer will point to a pre-existing node if there was one having the same value. This ensures that vertex points are properly shared by different partitions and compartments.

```
Main Menu
    N. New structure
    O. Open structure
    E. Edit structure
    S. Save structure
    Q. Quit program
Selection:
```

Listing 2. The Main Menu

The Main Menu provides a means of creating, opening, editing and saving structure files. There is also an option to quit the program.

When the program is run, the Main Menu (Listing 2) appears. At this point, you can create a new structure or open an existing one. This level also permits you to save the structure to a file and to exit the program.

```
Editing structure: SHADWELL/688.

Edit Structure Menu
    N. Name structure
    L. List compartments
    A. Add compartment
    E. Edit compartment
    D. Delete compartment
    X. Return to Main Menu
Selection:
```

Listing 3. The Edit Structure Menu

The Edit Structure Menu allows you to name the structure and add, edit or delete compartments. You can also view a list of existing compartments.

Once a valid structure exists, the "Edit Structure" command brings up the menu shown in Listing 3. The name of the current structure is displayed to reduce the possibility of mistakenly operating on the wrong one. There are commands for changing the structure name, listing the names (and serial numbers) of all existing compartments and for adding, editing and deleting compartments.

The compartment most recently added to the structure becomes the default compartment for editing. As seen in Listing 4, when the "Edit Compartment" command is issued and no compartment has previously been added or edited during the current editing session, then a list of existing compartments is displayed and the user is asked to chose one.

```
Current compartments:
Compartment 1 Laundry_Room
Compartment 2 Laundry_Passageway

ID of compartment to edit: 1

Editing structure: SHADWELL/688.
Editing compartment: 1 Laundry_Room.

Edit Compartment Menu
    S. Select compartment to edit
    N. Name compartment
    L. List partitions
    A. Add partition
    E. Edit partition
    D. Delete partition
    X. Return to Edit Structure Menu
Selection:
```

Listing 4. The Edit Compartment Menu

The Edit Compartment Menu has options for selecting and naming compartments and for adding, editing and deleting partitions.

Subsequently, the last-selected compartment will be the default, but there is a command to change that selection when desired. As a reminder of which compartment is currently being edited, the names of the structure and the selected compartment are displayed above the menu. The partitions which belong to the compartment may be listed, edited or deleted and new ones may be added.

Listing 5 shows the "Edit Partition Menu." As in the previous case, the program asks the user to select a partition to edit if none has been added or edited during the current session. Once selected, that partition will be the default until another selection is made. The current structure, compartment and partition are identified above the menu for reference.

The commands are similar to those discussed for the "Edit Structure" and "Edit Compartment" menus, except that the "Add" and "Delete" commands behave slightly differently. As was mentioned previously, it is important that the vertices be listed in the order in which they are encountered when the partition's perimeter is traversed. Therefore, when adding vertex points, the program keeps asking for vertex coordinates until the user closes the loop by entering the coordinates of the first point again. For verification, the program displays each of the coordinates, and the corresponding vertex serial number, as the points are entered.

It is not possible to edit any given vertex. Instead, the deletion command clears the entire list and you must reenter the coordinates of all of the partition's vertices. This was done for simplicity, to avoid the problems that can occur if points are deleted and inserted from the middle of the list. A more "user-unfriendly" vertex editor would allow points to be moved while maintaining their connections to adjacent vertices.

```
Current partitions:
Partition 1 Aft
Partition 2 Port
Partition 3 Fwd
Partition 4 Stbd
Partition 5 Deck
Partition 6 Ovhd

ID of partition to edit: 1

Editing structure: SHADWELL/688.
Editing compartment: 1 Laundry_Room.
Editing partition: 1 Aft.

Edit Partition Menu
    S. Select partition to edit
    N. Name partition
    L. List vertices
    A. Add vertices
    D. Delete vertices
    X. Return to Edit Compartment Menu
Selection:
```

Listing 5. The Edit Partition Menu

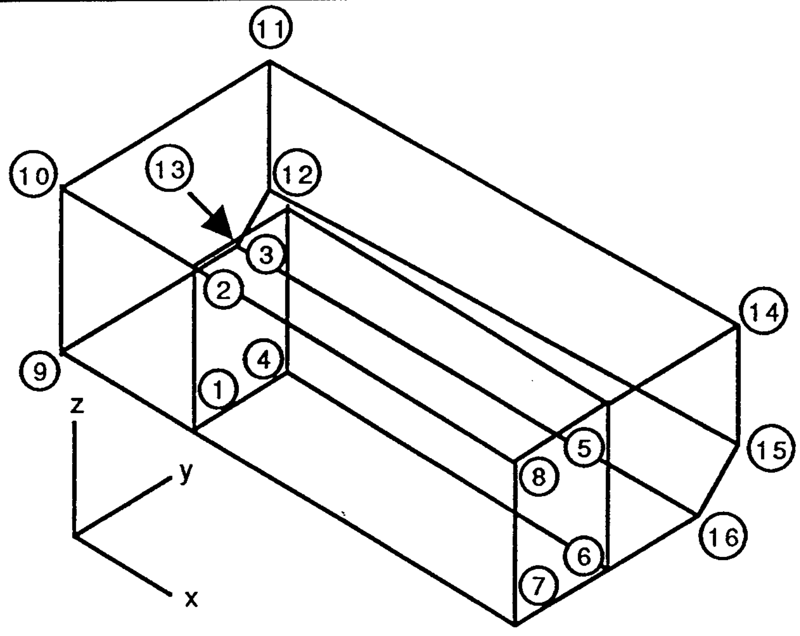
The Edit Partition Menu permits selection and naming of partitions and adding to or deleting the vertex list for a partition. To ensure that correct vertex order is maintained, individual points may not be edited or deleted.

3.5 SHADWELL/688 Example

As a demonstration of the application of the SAFE geometry concept, we have used the prototype data entry program to create a representation of the same SHADWELL/688 Laundry and Laundry Passageway area that was used for the CFAST example. Figure 16 illustrates the data entry sequence and includes the coordinates of the vertices. Note that there is nothing special about this particular sequence — the underlying uUDGraph ensures that the connections will be self-consistent regardless of the order in which partitions are entered. Also, within a partition, the starting point and direction in which the perimeter is traversed are both arbitrary.

Using the vertex coordinates from the figure, the data entry program produces the output file shown in Listing 6. The format uses keywords to identify the various parts of the structure. At present, these are for the benefit of users, making it easier to read the file. However, it is envisioned that the keywords will eventually be used by the input functions for error checking. For example, if there are supposed to be six compartments and only one "Compartment:" keyword is found, then some sort of error message could be generated. For the purposes of this demonstration, no such error checking has been implemented.

The "Structure;" keyword identifies the start of the data. The first two numbers after the keyword specify the highest compartment serial number previously used and the total number of compartments. This ensures that, if the data are read in and new compartments added, the additions will always receive unique serial numbers. Note that serial numbers are never reused so, if compartments have been deleted, the number of compartments may not match the serial number counter. The next two numbers (currently, always zeros) are place holders for planned future improvements and the line ends with the structure name.



Vertex Index	Coordinates	Vertex Index	Coordinates
1	(2.44, 0.00, 0.00)	2	(2.44, 0.00, 2.57)
3	(2.44, 1.75, 2.57)	4	(2.44, 1.75, 0.00)
5	(8.51, 1.75, 2.57)	6	(8.51, 1.75, 0.00)
7	(8.51, 0.00, 0.00)	8	(8.51, 0.00, 2.57)
9	(0.00, 0.00, 0.00)	10	(0.00, 0.00, 2.57)
11	(0.00, 3.86, 2.57)	12	(0.00, 3.86, 0.57)
13	(0.00, 3.28, 0.00)	14	(8.51, 4.15, 2.57)
15	(8.51, 4.15, 0.74)	16	(8.51, 3.43, 0.00)

Figure 16. SHADWELL/688 Laundry Room and Laundry Passageway Vertex Coordinates

The numbers in circles indicate the order in which vertex coordinates were entered. For each partition, the points must be in sequence, but the order of the partitions and the direction around the perimeter of each partition is arbitrary. The connections among the points will be internally consistent regardless of the numbering scheme. Coordinates (in meters) for each vertex are listed below the figure. Compare this diagram with the corresponding output file in Listing 6.

The beginning of data for each compartment is flagged by the "Compartment" keyword, which is followed by the compartment serial number, largest previously-used partition serial number, number of partitions and compartment name. Again, the maximum serial number may not be the same as the actual number of compartments. It should be noted that, at present, the partition serial numbers are only unique to the compartment so partitions belonging to different compartments can have the same serial number. This is not expected to be a problem, because the structure of the data file inherently associates a partition list with a specific compartment.

```
Structure: 2 2 0 0 SHADWELL/688
Compartment: 1 6 6 Laundry_Room
    Partition: 1 4 Aft
        [00000001] [00000002] [00000003] [00000004]
    Partition: 2 4 Port
        [00000004] [00000003] [00000005] [00000006]
    Partition: 3 4 Fwd
        [00000007] [00000008] [00000005] [00000006]
    Partition: 4 4 Stbd
        [00000001] [00000002] [00000008] [00000007]
    Partition: 5 4 Deck
        [00000001] [00000004] [00000006] [00000007]
    Partition: 6 4 Ovhd
        [00000002] [00000003] [00000005] [00000008]
Compartment: 2 9 9 Laundry_Passageway
    Partition: 1 5 Aft
        [00000009] [00000010] [00000011] [00000012] [00000013]
    Partition: 2 4 Port_Upper
        [00000012] [00000011] [00000014] [00000015]
    Partition: 3 4 Port_Lower
        [00000013] [00000012] [00000015] [00000016]
    Partition: 4 5 Fwd
        [00000006] [00000005] [00000014] [00000015] [00000016]
    Partition: 5 4 Stbd_Fwd
        [00000004] [00000003] [00000005] [00000006]
    Partition: 6 4 Mid
        [00000001] [00000002] [00000003] [00000004]
    Partition: 7 4 Stbd_Aft
        [00000009] [00000010] [00000002] [00000001]
    Partition: 8 6 Deck
        [00000009] [00000013] [00000016] [00000006] [00000004]
        [00000001]
    Partition: 9 6 Ovhd
        [00000010] [00000011] [00000014] [00000005] [00000003]
        [00000002]
VertexGraph: 16 16
GNode: 1 <2.44, 0, 0> 12
    [00000002] [00000004] [00000002] [00000007] [00000004]
    [00000007] [00000002] [00000004] [00000002] [00000009]
    [00000004] [00000009]
GNode: 2 <2.44, 0, 2.57> 12
    [00000001] [00000003] [00000001] [00000008] [00000003]
    [00000008] [00000001] [00000003] [00000010] [00000001]
    [00000003] [00000010]
```

Listing 6. File for SAFE-Compatible SHADWELL/688 Example

The actual geometry of the Laundry Room/Laundry Passageway area is represented in this data file. Compare with Listing 1 and note how much information is ignored by CFAST.

```
GNode: 3 <2.44, 1.75, 2.57> 12
     [00000002] [00000004] [00000004] [00000005] [00000002]
     [00000005] [00000004] [00000005] [00000002] [00000004]
     [00000005] [00000002]
GNode: 4 <2.44, 1.75, 0> 12
     [00000003] [00000001] [00000003] [00000006] [00000001]
     [00000006] [00000003] [00000006] [00000003] [00000001]
     [00000006] [00000001]
GNode: 5 <8.51, 1.75, 2.57> 12
     [00000003] [00000006] [00000008] [00000006] [00000003]
     [00000008] [00000006] [00000014] [00000003] [00000006]
     [00000014] [00000003]
GNode: 6 <8.51, 1.75, 0> 12
     [00000005] [00000004] [00000005] [00000007] [00000004]
     [00000007] [00000005] [00000016] [00000005] [00000004]
     [00000016] [00000004]
GNode: 7 <8.51, 0, 0> 6
     [00000008] [00000006] [00000008] [00000001] [00000006]
     [00000001]
GNode: 8 <8.51, 0, 2.57> 6
     [00000007] [00000005] [00000002] [00000007] [00000005]
     [00000002]
GNode: 9 <0, 0, 0> 6
     [00000010] [00000013] [00000010] [00000001] [00000013]
     [00000001]
GNode: 10 <0, 0, 2.57> 6
     [00000009] [00000011] [00000009] [00000002] [00000011]
     [00000002]
GNode: 11 <0, 3.86, 2.57> 6
     [00000010] [00000012] [00000012] [00000014] [00000010]
     [00000014]
GNode: 12 <0, 3.86, 0.57> 6
     [00000011] [00000013] [00000011] [00000015] [00000013]
     [00000015]
GNode: 13 <0, 3.28, 0> 6
     [00000012] [00000009] [00000012] [00000016] [00000009]
     [00000016]
GNode: 14 <8.51, 4.15, 2.57> 6
     [00000011] [00000015] [00000005] [00000015] [00000011]
     [00000005]
GNode: 15 <8.51, 4.15, 0.74> 6
     [00000014] [00000012] [00000012] [00000016] [00000014]
     [00000016]
GNode: 16 <8.51, 3.43, 0> 6
     [00000015] [00000013] [00000015] [00000006] [00000013]
     [00000006]
```

Listing 6. File for SAFE-Compatible SHADWELL/688 Example (Continued)

Each partition is identified by the "Partition" keyword. The partition serial number, number of vertex points and partition name are given, followed by a list of the vertex serial numbers. Note that the same vertex serial numbers appear in multiple partitions because, as was discussed above, the vertices are shared. For example, in compartment 1, each of the eight vertices appears three times, once in each of three different partitions.

The graph containing the master list of vertex points is written after the last partition of the last compartment, using the "VertexGraph:" keyword. As usual, the maximum vertex serial number and current number of vertex points in the graph are given. Individual graph nodes are identified by the "GNode:" keyword and each entry includes the vertex serial number, the vector coordinates of the point associated with the vertex, the number of connections and a list of serial numbers of the connected nodes. Since each connection represents an edge and edges, like vertices, are shared by different partitions, connections can appear more than once within a node entry.

4.0 CONCLUSIONS AND RECOMMENDATIONS

This work has demonstrated the feasibility of representing complex, real-world geometries in a format that could be used by an advanced fire model. Comparison of Listing 1 (the geometry portion of a CFAST input file for the SHADWELL/688 laundry/laundry passageway) and Listing 6 (the same area represented in the prototype SAFE format) gives an idea of the amount of information which is ignored by the current generation of zone fire models. Potentially, that information could be used to improve the accuracy of zone model predictions or to extend the range of scenarios over which predictions can be made.

However, to realize that potential, significant advances must be made in the fire model itself. Unfortunately, this is not simply a matter of changing the input functions so that CFAST "understands" a new file format. The limiting assumptions, such as those regarding the shapes of compartments, are deeply imbedded in CFAST — they appear at every point that an area or volume is calculated, for example.

Because of the difficulties inherent in changing the foundations of a structure without causing it to collapse, it might be easier and more cost effective in the long run to develop a new, next-generation zone model than to continue to adapt CFAST indefinitely. Clearly, this would be a long-term project and CFAST will continue to be useful for many years to come. However, it is recommended that consideration be given to starting such a project.

Further work will also be required to add important features to the geometry model. For example, the proof-of-concept version reported here does not include vents, such as doors and hatches, or mechanical ventilation, both of which are supported by CFAST. However, these refinements should be relatively easy to add, since the entire SAFE concept was designed to make such modifications as simple as possible. Vents, for instance, would most likely be added by a two-step process:

a. creating a new pVent class, which would encapsulate the properties and behaviors of vents (*e.g.* location, shape, area and flow restriction); and

b. modifying the pPartition class to include a list of all vents associated with a particular partition and to add methods for accessing vent-related information.

Another area in which further work will be needed is the user interface. As was noted above, the current demonstration data entry program is very crude. We envision that, for an actual fire model, the input would be graphical, so the user could simply drag a point to a new location to move a vertex or call up a dialog box to allow information about a specific compartment to be entered. Some user-interaction investigations would be useful to assist in designing an interface that would be truly user friendly but, for the most part, this part of the project would be an exercise in applying known methods to a new problem.

5.0 ACKNOWLEDGMENTS

We wish to acknowledge the helpful comments and suggestions provided by Jean Bailey (NRL Code 6180) and Alex Maranghides (GeoCenters).

The work described in this report was performed by the Chemistry Division of the Materials Science and Component Technology Directorate, Naval Research Laboratory. The work was funded by the Office of Naval Research, Code 334, under the Damage Control Task of the FY 1998 Surface Ship Hull, Mechanical, and Electrical Technology Program (PE0602121N).

6.0 REFERENCES

1 J.A. Rocket, personal communication, January 1995.

2 H.E. Mitler and J.A. Rocket, "User's Guide to FIRST, A Comprehensive Single-Room Fire Model," NIST NBSIR 87-3595, September 1987.

3 J.A. Rocket, personal communication, January 1995.

4 R.D. Peacock, G.P. Forney, P. Reneke, R. Portier and W.W. Jones, "CFAST, the Consolidated Model of Fire Growth and Smoke Transport," NIST Tech Note 1299, February 1993.

5 R.D. Peacock, P. Reneke, W.W. Jones, R.W. Bukowski and G.P. Forney, "A User's Guide to FIRST: Engineering Tools for Estimating Fire Growth and Smoke Transport," NIST Special Publication 921, October 1997.

6 J.B. Hoover, J.L. Bailey and P.A. Tatem, "An Improved Radiation Transport Submodel for CFAST," Combust. Sci. and Tech., 127, p. 213, 1997.

7 H. Schildt, "Teach Yourself C++," Osborne McGraw-Hill, 1992 .

8 P. Anderson and G. Anderson, "Navigating C++ and Object-Oriented Design," Prentice Hall, 1998.

APPENDIX A STRUCTURE CLASS

A.1 pStructure Capabilities

The pStructure class represents the geometry of the entire modeling domain. This class includes methods for:

a. creation of an empty structure;

b. copying of an existing structure;

c. reading and writing the structure name;

d. finding an existing vertex by its value or serial number;

e. iterating through the vertex points;

f. counting the number of vertex points in the graph and testing for an empty graph;

g. creating, adding and deleting compartments;

h. finding a compartment by its serial number;

i. iterating through the list of compartments;

j. counting the number of compartments and testing for an empty compartment list; and

k. structure input and output.

NOTE: Vertices are added and deleted via the pPartition interface, therefore there are no AddVertex or DeleteVertex functions in pStructure.

A.2 pStructure API

<u>pStructure</u>

Purpose: Class constructor. Syntax 1 is the default constructor; syntax 2 is the copy constructor. For syntax 2, the structure reference parameter is const.

Syntax:
```
1    pStructure()
2    pStructure(Structure&)
```

Parameters: `pStructure&` Reference to the existing pStructure object which is to be copied.

Return value: None.

<u>~pStructure</u>

Purpose: Class destructor.

Syntax:	1	`~pStructure()`	
Parameters:			None.
Return value:			None.

pStructure::Name

Purpose: Syntax 1 returns the name of the structure; syntax 2 returns the current name and sets a new name. For syntax 2, the string reference parameter is const.

Syntax:	1	`string`	`Name()`
	2	`string`	`Name(String&)`
Parameters:	`string&`		Reference to a string containing the new structure name.
Return value:			String containing the structure name at the time the function was called.

pStructure::FindVertex

Purpose: Searches for a pVertex, within the current pStructure, which matches the criteria. Syntax 1 searches by object value; syntax 2 searches by object serial number.

Syntax:	1	`pVertexPtr`	`FindVertex(p3DPoint&)`
	2	`pVertexPtr`	`FindVertex(long)`
Parameters:	`p3DPoint&`		Reference to a p3DPoint containing the value which is to be found.
	`long`		Serial number of the pVertex object which is to be found.
Return value:			Pointer to the found pVertex object; nil if no object was found.

pStructure::VertexBegin

Purpose: Sets a list iterator to the first element of the vertex master list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

Syntax:	1	`pVertexItr`	`VertexBegin()`
	2	`pVertexConstItr`	`VertexBegin()`
Parameters:			None.
Return value:			Iterator which refers to the first vertex in the master list.

pStructure::VertexEnd

Purpose: Sets a list iterator to past the last element of the vertex master list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

Syntax:
```
1    pVertexItr         VertexEnd()
2    pVertexConstItr    VertexEnd()
```

Parameters: None.

Return value: Iterator which refers to one element past the last vertex in the master list.

pStructure::VertexEmpty

Purpose: Returns the state of the master vertex list.

Syntax:
```
1    bool    VertexEmpty()
```

Parameters: None.

Return value: True if the master vertex list is empty; false if it is not.

pStructure::VertexSize

Purpose: Returns the size of (number of elements in) the master vertex list.

Syntax:
```
1    int     VertexSize()
```

Parameters: None.

Return value: Number of vertices in the master vertex list.

pStructure::AddCompartment

Purpose: Creates a new pCompartment object and adds it to the compartment list.

Syntax:
```
1    pCmptPtr   AddCompartment()
```

Parameters: None.

Return value: Pointer to the new compartment; nil if no compartment was created.

pStructure::DeleteCompartment

Purpose: Deletes an existing pCompartment object from the compartment list and frees the associated memory. All partitions owned by the deleted compartment are also deleted.

Syntax:
```
1    void    DeleteCompartment(pCmptPtr)
```

Parameters:	`pCmptPtr`	Pointer to the pCompartment object which is to be deleted.
Return value:		None.

<u>pStructure::FindCompartment</u>

Purpose: Searches for a pCompartment, within the current pStructure, which matches the specified compartment serial number.

```
Syntax:     1    pCmptPtr   FindCompartment(long)
```

Parameters:	`long`	Serial number of the pCompartment object which is to be found.
Return value:		Pointer to the found pCompartment object; nil if no object was found.

<u>pStructure::CmptBegin</u>

Purpose: Sets a list iterator to the first element of the compartment list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

```
Syntax:     1    pCmptItr        CmptBegin()
            2    pCmptConstItr   CmptBegin()
```

Parameters:		None.
Return value:		Iterator which refers to the first compartment in the list.

<u>pStructure::CmptEnd</u>

Purpose: Sets a list iterator to past the last element of the compartment list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

```
Syntax:     1    pCmptItr        CmptEnd()
            2    pCmptConstItr   CmptEnd()
```

Parameters:		None.
Return value:		Iterator which refers to one element past the last compartment in the list.

<u>pStructure::CmptEmpty</u>

Purpose: Returns the state of the compartment list.

```
Syntax:     1    bool   CmptEmpty()
```

Parameters:		None.

Return value:		True if the compartment list is empty; false if it is not.

pStructure::CmptSize

Purpose:	Returns the size of (number of elements in) the compartment list.	
Syntax:	1 `int`	`CmptSize()`
Parameters:		None.
Return value:		Number of compartments in the compartment list.

pStructure::Operator =

Purpose:	Assigns value of one pStructure variable to another pStructure variable. The right hand side structure reference is const.	
Syntax:	1 `pStructure& = pStructure&`	
Parameters:	`pStructure&`	Reference to a source pStructure variable.
Return value:		Reference to a destination pStructure variable.

pStructure::Operator <<

Purpose:	Inserts pStructure into output stream. The structure reference is const.	
Syntax:	1 `ostream& << pStructure&`	
Parameters:	`pStructure&`	Reference to a source pStructure variable.
Return value:		Reference to an output stream.

pStructure::Operator >>

Purpose:	Extracts pStructure from input stream.	
Syntax:	1 `ostream& >> pStructure&`	
Parameters:	`pStructure&`	Reference to a destination pStructure variable.
Return value:		Reference to an input stream.

APPENDIX B COMPARTMENT CLASS

B.1 pCompartment Capabilities

The pCompartment class represents a space of arbitrary shape within the structure. For the purposes of the SAFE geometry model, hallways, stairways and similar spaces are considered to be compartments. Functions and operators for the pCompartment class include:

a. reading the compartment serial number;

b. reading and writing the name of the compartment;

c. creating, adding and deleting partitions;

d. finding an existing partition by serial number;

e. iterating through the list of connections;

f. counting the number of connections and testing for empty connection lists;

g. compartment input and output.

NOTE: Compartments are created and destroyed via the pStructure interface, therefore there are no public constructor or destructor functions in pCompartment.

B.2 pCompartment API

<u>pCompartment::Parent</u>

Purpose:	Returns a pointer to the parent pStructure object.
Syntax:	`1 pStructPtr Parent()`
Parameters:	None.
Return value:	Pointer to the pStructure object which contains the current pCompartment object.

<u>pCompartment::ID</u>

Purpose:	Returns the serial number of the current compartment.
Syntax:	`1 long ID()`
Parameters:	None.
Return value:	Serial number of the current pCompartment object.

pCompartment::Name

Purpose: Syntax 1 returns the name of the compartment; syntax 2 returns the current name and sets a new name. For syntax 2, the string reference parameter is const.

Syntax:
```
1    string    Name()
2    string    Name(const string&)
```

Parameters: `string&` Reference to a string containing the new compartment name.

Return value: String containing the compartment name at the time the function was called.

pCompartment::AddPartition

Purpose: Creates a new pPartition object and adds it to the partition list.

Syntax:
```
1    pPartPtr    AddPartition()
```

Parameters: None.

Return value: Pointer to the new partition; nil if no partition was created.

pCompartment::DeletePartition

Purpose: Deletes an existing pPartition object from the partition list and frees the associated memory. Vertices shared by the deleted partition are deleted only if they are not used by another partition.

Syntax:
```
1    void    DeletePartition(pPartPtr)
```

Parameters: `pPartPtr` Pointer to the pPartition object which is to be deleted.

Return value: None.

pCompartment::FindPartition

Purpose: Searches for a pPartition, within the current pCompartment, which matches the specified partition serial number.

Syntax:
```
1    pPartPtr    FindPartition(long)
```

Parameters: `long` Serial number of the pPartition object which is to be found.

Return value: Pointer to the found pPartition object; nil if no object was found.

pCompartment::begin

Purpose: Sets a list iterator to the first element of the partition list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

Syntax:
```
1     pPartItr        begin()
2     pPartConstItr   begin()
```

Parameters: None.

Return value: Iterator which refers to the first partition in the list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

pCompartment::end

Purpose: Sets a list iterator to past the last element of the partition list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

Syntax:
```
1     pPartItr        end()
2     pPartConstItr   end()
```

Parameters: None.

Return value: Iterator which refers to one element past the last partition in the list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

pCompartment::empty

Purpose: Returns the state of the partition list.

Syntax:
```
1     bool            empty()
```

Parameters: None.

Return value: True if the partition list is empty; false if it is not.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

pCompartment::size

Purpose: Returns the size of the partition list.

Syntax:
```
1     int             size()
```

Parameters: None.

Return value:		Number of partitions in the partition list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

<u>pCompartment::Operator <<</u>

Purpose: Inserts pCompartment into output stream. The compartment reference is const.

Syntax: 1 `ostream& << pCompartment&`

Parameters:	`pCompartment&`	Reference to a source pCompartment variable.
Return value:		Reference to an output stream.

<u>pCompartment::Operator >></u>

Purpose: Extracts pCompartment from input stream.

Syntax: 1 `ostream& >> pCompartment&`

Parameters:	`pCompartment`	Reference to a destination pCompartment variable.
Return value:		Reference to an input stream.

APPENDIX C PARTITION CLASS

C.1 pPartition Capabilities

The pPartition class represents the bounding planes of a compartment. A pPartition object is approximately equivalent to a wall (bulkhead), floor (deck) or ceiling (overhead). Class methods are available for performing the following operations:

a. reading the partition serial number;

b. reading and writing the partition name;

c. creating, adding and deleting vertex points;

d. finding an existing vertex by value or by serial number;

e. connecting and disconnecting pairs of existing vertex points;

f. iterating through the list of vertex points;

g. counting the number of vertex points and testing for empty vertex lists; and

h partition input and output.

NOTE: Partitions are created and destroyed via the pCompartment interface, therefore there are no public constructor or destructor functions in pPartition.

C.2 pPartition API

pPartition::Parent

Purpose:	Returns a pointer to the parent pCompartment object.
Syntax:	1 `pCmptPtr` `Parent()`
Parameters:	None.
Return value:	Pointer to the pCompartment object which contains the current pPartition object.

pPartition::ID

Purpose:	Returns the serial number of the current partition.
Syntax:	1 `long` `ID()`
Parameters:	None.
Return value:	Serial number of the current pPartition object.

pPartition::Name

Purpose: Syntax 1 returns the name of the partition; syntax 2 returns the current name and sets a new name. For syntax 2, the string reference parameter is const.

Syntax:

```
1    string    Name()
2    string    Name(string&)
```

Parameters:	`string&`	Reference to a string containing the new partition name.
Return value:		String containing the partition name at the time the function was called.

pPartition::AddVertex

Purpose: Creates a new pVertex object, adds it to both the structure's master vertex list and the partition's local vertex list and returns a pointer to the vertex. Syntax 1 creates an unconnected vertex; syntax 2 creates a vertex connected to a previous vertex.

Syntax:

```
1    pVertexPtr    AddVertex(p3DPoint&)
2    pVertexPtr    AddVertex(p3DPoint&,
                   pVertexPtr)
```

Parameters:	`p3DPoint&`	Reference to a vector representing the coordinates of the new point.
	`pVertexPtr`	Pointer to a previous vertex to which the new vertex will be connected.
Return value:		Pointer to the new partition; nil if no partition was created. See Note 2.

NOTE 1: Duplicate entries are not created in the master list. If the value of an existing vertex is the same as that of the p3DPoint& parameter, a pointer to the existing object is returned.

NOTE 2: Only the initial point in the list may be unconnected. All points after the initial one must have a connection to the preceding point in the existing list. If either of these constraints is violated, the point will not be added and nil will be returned.

pPartition::DeleteVertices

Purpose: Deletes all existing pVertex objects from the partition's vertex list. Vertices are deleted from the master list only if they are not in use by another partition.

Syntax:

```
1    void    DeleteVertices()
```

Parameters: None.

Return value:	None.

pPartition::FindVertex

Purpose:	Searches for a pVerterx, within the current pPartition, which matches the specified partition serial number.	
Syntax:	1 `pVertexPtr`	`FindVertex(p3DPoint&)`
	2 `pVertexPtr`	`FindVertex(long)`
Parameters:	`p3DPoint&`	Reference to a p3DPoint object containing the value which is to be found.
	`long`	Serial number of the pPartition object which is to be found.
Return value:		Pointer to the found pPartition object; nil if no object was found.

NOTE: pPartition::FindVertex only searches the vertex list within the partition; it does not search the structure's master vertex list. To search the master list, use pStructure::FindVertex .

pPartition::begin

Purpose:	Sets a list iterator to the first element of the partition's vertex list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.	
Syntax:	1 `pVertexItr`	`begin()`
	2 `pVertexConstItr`	`begin()`
Parameters:		None.
Return value:		Iterator which refers to the first partition in the list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

pPartition::end

Purpose:	Sets a list iterator to past the last element of the partition's vertex list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.	
Syntax:	1 `pVertexItr`	`end()`
	2 `pVertexConstItr`	`end()`
Parameters:		None.
Return value:		Iterator which refers to one element past the last vertex in the partition's vertex list.

NOTE:	Lower case function name is used for consistency with STL nomenclature for container classes.

<u>pPartition::empty</u>

Purpose:	Returns state of the partition's vertex list.		
Syntax:	1	`bool`	`empty()`
Parameters:			None.
Return value:			True if the partition's vertex list is empty; false if it is not.

NOTE:	Lower case function name is used for consistency with STL nomenclature for container classes.

<u>pPartition::size</u>

Purpose:	Returns the size of the partition's vertex list.		
Syntax:	1	`int`	`size()`
Parameters:			None.
Return value:			Number of vertices in the partition's vertex list.

NOTE:	Lower case function name is used for consistency with STL nomenclature for container classes.

<u>pPartition::Operator <<</u>

Purpose:	Inserts pPartition into output stream. The partition reference is const.	
Syntax:	1 `ostream& << pPartition&`	
Parameters:	`pStructure&`	Reference to a source pPartition variable.
Return value:		Reference to an output stream.

<u>pPartition::Operator >></u>

Purpose:	Extracts pPartition from input stream.	
Syntax:	1 `ostream& >> pPartition&`	
Parameters:	`pStructure&`	Reference to a destination pPartition variable.
Return value:		Reference to an input stream.

APPENDIX D 3-D VECTOR CLASS

D.1 u3DVector Capabilities

Class u3DVector implements Cartesian vectors in three dimensions. It supports the following functions and operations:

a. vector magnitude;

b. vector normalization;

c. scalar multiplication;

d. scalar division;

e. dot product;

f. cross product;

g. addition;

h. subtraction;

i. logical equality;

j. logical inequality; and

k. vector I/O.

NOTE: The u3DVector class is declared as a C++ template of the form u3DVector<T>, where T is a generic variable type. In order to use the class, T must be instantiated as a specific class, such as int or float. Class <T> is required to have at least the following operators: multiplication, division, addition, subtraction, logical equality, logical inequality, stream insertion and stream extraction.

D.2 u3DVector API

u3DVector

Purpose:	Class constructor. Creates a 3-dimensional, Cartesian vector from three scalars of type T.	
Syntax:	1 u3DVector(T, T, T)	
Parameters:	T	Coordinate values in the order x, y, z.
Return value:		None.

NOTE: Default values for all three parameters are zero.

~u3DVector

Purpose:	Class destructor.		
Syntax:	1	~u3DVector()	
Parameters:			None.
Return value:			None.

u3DVector::x

Purpose: Syntax 1 returns the x-coordinate value; syntax 2 returns the current x-coordinate and sets a new x-coordinate.

Syntax:
```
1    T    x()
2    T    x(T)
```

Parameters:	T	New x-coordinate value.
Return value:		X-coordinate value at the time the function was called.

u3DVector::y

Purpose: Syntax 1 returns the y-coordinate value; syntax 2 returns the current y-coordinate and sets a new y-coordinate.

Syntax:
```
1    T    y()
2    T    y(T)
```

Parameters:	T	New y-coordinate value.
Return value:		Y-coordinate value at the time the function was called.

u3DVector::z

Purpose: Syntax 1 returns the z-coordinate value; syntax 2 returns the current z-coordinate and sets a new z-coordinate.

Syntax:
```
1    T    z()
2    T    z(T)
```

Parameters:	T	New z-coordinate value.
Return value:		Z-coordinate value at the time the function was called.

u3DVector::abs

Purpose: Returns the magnitude of the source vector. Source vector is const.

Syntax:
```
1    T    abs(u3DVector<T>&)
```

Parameters:	`u3DVector<T>&`	Reference to source vector instantiated for type T.
Return value:		Magnitude of source vector.

<u>u3DVector::normalize</u>

Purpose: Returns a unit vector parallel to the source vector. Source vector is const.

Syntax:
```
1      u3DVector<T>     normalize(u3DVector<T>&)
```

Parameters:	`u3DVector<T>&`	Reference to vector instantiated for type T.
Return value:		Unit vector, instantiated for type T.

<u>u3DVector::Operator *</u>

Purpose: Syntaxes 1 and 2 return the product of a scalar of type T multiplied by a vector instantiated for type T; syntax 3 returns the scalar (dot) product of two vectors instantiated for type T. Both the right- and left hand sides are const.

Syntax:
```
1      u3DVector<T>     T * u3DVector<T>&
2      u3DVector<T>     u3DVector<T>& * T
3      T                u3DVector<T>& * u3DVector<T>&
```

Parameters:	`T`	Scalar of type T.
	`u3DVector<T>&`	Reference to vector instantiated for type T.
Return value:		Product vector, instantiated for type T (syntaxes 1 and 2); scalar (dot) product, instantiated for type T (syntax 3).

<u>u3DVector::Operator /</u>

Purpose: Divides a vector instantiated for type T by a scalar of type T. Both the right- and left hand sides are const.

Syntax:
```
1      u3DVector<T>& / T
```

Parameters:	`u3DVector<T>&`	Reference to vector instantiated for type T.
	`T`	Scalar of type T.
Return value:		Quotient vector, instantiated for type T.

<u>u3DVector::Operator ^</u>

Purpose: Returns the vector (cross) product of two vectors instantiated for type T. Both the right- and left hand sides are const.

Syntax:
```
1      u3DVector<T>& ^ u3DVector<T>&
```

Parameters:	`u3DVector<T>&`	Reference to vector instantiated for type T.

Return value: Vector (cross) product , instantiated for type T.

u3DVector::Operator +

Purpose: Returns the sum of two vectors instantiated for type T. Both the right- and left hand sides are const.

Syntax: 1 `u3DVector<T>& + u3DVector<T>&`

Parameters: `u3DVector<T>&` Reference to vector instantiated for type T.

Return value: Sum vector , instantiated for type T.

u3DVector::Operator -

Purpose: Returns the difference of two vectors instantiated for type T. Both the right- and left hand sides are const.

Syntax: 1 `u3DVector<T>& - u3DVector<T>&`

Parameters: `u3DVector<T>&` Reference to vector instantiated for type T.

Return value: Difference vector, instantiated for type T.

u3DVector::Operator ==

Purpose: Compares components of two vectors for logical equality. Both the right- and left hand sides are const.

Syntax: 1 `u3DVector<T>& == u3DVector<T>&`

Parameters: `u3DVector<T>&` Reference to vector instantiated for type T.

Return value: True if components of right- and left hand sides are equal; false if they are not.

u3DVector::Operator !=

Purpose: Compares components of two vectors for logical inequality. Both the right- and left hand sides are const.

Syntax: 1 `u3DVector<T>& != u3DVector<T>&`

Parameters: `u3DVector<T>&` Reference to vector instantiated for type T.

Return value: True if components of right- and left hand sides are not equal; false if they are.

u3DVector::Operator <<

Purpose: Inserts vector into output stream. The vector reference is const.

Syntax:	1 `ostream& << u3DVector<T>&`	
Parameters:	`u3DVector<T>&`	Reference to a source vector instantiated for type T.
Return value:		Reference to an output stream.

u3DVector::Operator >>

Purpose:	Extracts vector from input stream. The vector reference is const.	
Syntax:	1 `istream& >> u3DVector<T>&`	
Parameters:	`u3DVector<T>&`	Reference to a destination vector instantiated for type T.
Return value:		Reference to an input stream.

APPENDIX E CYCLIC, UNDIRECTED GRAPH CLASS

E.1 Graph Capabilities

NOTE: Nodes are created and destroyed via the uUDGraph interface, therefore there are no public constructor or destructor functions in uUDGraphNode.

Collectively, the classes uUDGraph and uUDGraphNode implement the mathematical concept of a cyclic, undirected graph. Only one copy of a graph node having a given value is permitted, but multiple connections between pairs of nodes are allowed. The uUDGraph class provides the following capabilities:

a. construction of an empty graph object;

b. copying of an existing graph;

c. adding a new node, with or without a connection to an existing node;

d. deleting an existing node;

e. finding an existing node by its value or serial number;

f. determining whether a node pointer points to a member of the current graph;

g. adding new connections between two existing nodes;

h. deleting connections between two nodes;

i. iterating through the list of nodes;

j. counting the number of nodes in the list and testing for empty node lists;

k. graph equality and inequality comparisons; and

l. graph input and output.

In addition to the above, the following capabilities are provided by uUDGraphNode class functions and operators:

a. reading the node serial number;

b. reading and writing the value of the node;

c. iterating through the list of connections;

d. counting the number of connections and testing for empty connection lists;

e. graph node equality and inequality comparisons; and

f. graph node input and output.

E.2 uUDGraph API

uUDGraph

Purpose: Class constructor. Syntax 1 is the default constructor; syntax 2 is the copy constructor. For syntax 2, the graph reference parameter is const.

Syntax:
```
1       uUDGraph()
2       uUDGraph(uUDGraph&)
```

Parameters: `uUDGraph&` Reference to the existing pStructure object which is to be copied.

Return value: None.

~uUDGraph

Purpose: Class destructor

Syntax:
```
1       ~uUDGraph()
```

Parameters: None.

Return value: None.

uUDGraph::AddNode

Purpose: Creates a new uUDGraphNode object, adds it to the graph's node list and returns a pointer to the node. Syntax 1 creates an unconnected node; syntax 2 creates a node connected to a previous node.

Syntax:
```
1       uUDGraphNodePtr     AddNode(p3DPoint&)
2       uUDGraphNodePtr     AddNode(p3DPoint&,
                                    uUDGraphNodePtr)
```

Parameters:	
`p3DPoint&`	Reference to a vector representing the coordinates of the new point.
`uUDGraphNodePtr`	Pointer to a previous node to which the new node will be connected.

Return value: Pointer to the new node; nil if no node was created.

NOTE: Duplicate nodes are not created. If the value of an existing node is the same as that of the p3DPoint& parameter, a pointer to the existing object is returned.

uUDGraph::DeleteNode

Purpose: Deletes an existing node.

Syntax:
```
1       void        DeleteNode(uUDGraphNodePtr)
```

Parameters: `uUDGraphNodePtr` Pointer to the uUDGraphNode object which is to be deleted.

Return value: None.

uUDGraph::FindNode

Purpose: Searches for a uUDGraphNode, within the current uUDGraph, which matches the criteria. Syntax 1 searches by object value; syntax 2 searches by object serial number.

Syntax:

```
1    uUDGraphNodePtr    FindNode(p3DPoint&)
2    uUDGraphNodePtr    FindNode(long)
```

Parameters:	`p3DPoint&`	Reference to a p3DPoint object containing the value which is to be found.
	`long`	Serial number of the uUDGraphNode object which is to be found

Return value: Pointer to the found uUDGraphNode object; nil if no object was found.

uUDGraph::IsGraphNode

Purpose: Tests graph node for membership in the current graph. The node pointer is const.

Syntax:

```
1    bool    IsGraphNode(uUDGraphNodePtr)
```

Parameters: `uUDGraphNodePtr` Pointer to the uUDGraphNode object which is to be tested.

Return value: True if the graph node is a member of the current graph; false if it is not.

uUDGraph::IsConnected

Purpose: Tests whether two graph nodes are directly connected. Both node pointers are const.

Syntax:

```
1    bool    IsConnected(uUDGraphNodePtr,
                     uUDGraphNodePtr)
```

Parameters: `uUDGraphNodePtr` Pointer to one of the uUDGraphNode objects which are to be tested.

Return value: True if the nodes are directly connected; false if they are not.

uUDGraph::begin

Purpose: Sets a list iterator to the first element of the graph's node list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

Syntax:	1	`uUDGraphNodeItr`	`begin()`
	2	`uUDGraphNodeConstItr`	`begin()`

Parameters: None.

Return value: Iterator which refers to the first partition in the list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

uUDGraph::end

Purpose: Sets a list iterator to past the last element of the partition's vertex list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

Syntax:	1	`uUDGraphNodeItr`	`end()`
	2	`uUDGraphNodeConstItr`	`end()`

Parameters: None.

Return value: Iterator which refers to one element past the last node in the graph's node list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

uUDGraph::empty

Purpose: Returns state of the graph's node list.

Syntax: 1 `bool` `empty()`

Parameters: None.

Return value: True if the graph's node list is empty; false if it is not.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

uUDGraph::size

Purpose: Returns the size of the graph's node list.

Syntax: 1 `int` `size()`

Parameters: None.

Return value: Number of nodes in the graph's node list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

uUDGraph::erase

Purpose:	Erases element(s) from the graph's node list. Syntax 1 erases a single node; syntax 2 erases a range of nodes.	

Syntax:
```
1    void     erase(uUDGraphNodeItr)
2    void     erase(uUDGraphNodeItr,
                    uUDGraphNodeItr)
```

Parameters:	`uUDGraphNodePtr`	Pointer to the node which is to be erased (syntax 1); pointer to the first and last nodes in the range which is to be erased (syntax 2).
Return value:		None.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

uUDGraph::AddConnection

Purpose: Creates a new connection between two nodes if both exist and are members of the same graph.

Syntax:
```
1    bool     AddConnection(uUDGraphNodePtr,
                    uUDGraphNodePtr)
```

Parameters:	`uUDGraphNodePtr`	Pointer to one of the two nodes which are to be connected.
Return value:		True if connection was made, false if it was not.

uUDGraph::DeleteConnection

Purpose: Deletes a connection between two nodes if both exist, are members of the same graph and were previously connected.

Syntax:
```
1    bool     DeleteConnection(uUDGraphNodePtr,
                    uUDGraphNodePtr)
```

Parameters:	`uUDGraphNodePtr`	Pointer to the one of the two nodes which are to be disconnected.
Return value:		True if connection was deleted, false if it was not.

uUDGraph::Operator =

Purpose: Assigns value of one uUDGraph variable to another uUDGraph variable. The right hand side graph reference is const.

Syntax:
```
1    uUDGraph& = uUDGraph&
```

Parameters:	`uUDGraph&`	Reference to a source uUDGraph variable
Return value:		Reference to a destination uUDGraph variable.

uUDGraph::Operator <<

Purpose:	Inserts uUDGraph into output stream. The graph reference is const.	
Syntax:	1 `ostream&` << uUDGraph&	
Parameters:	`uUDGraph&`	Reference to a source uUDGraph variable.
Return value:		Reference to an output stream

uUDGraph::Operator >>

Purpose:	Extracts uUDGraph from input stream.	
Syntax:	1 `ostream&` >> uUDGraph&	
Parameters:	`uUDGraph&`	Reference to a destination uUDGraph variable.
Return value:		Reference to an input stream.

uUDGraph::Operator ==

Purpose:	Compares two graphs for logical equality. Both the right- and left hand sides are const.	
Syntax:	1 `bool`	`uUDGraph& == uUDGraph&`
Parameters:	`uUDGraph&`	Reference to graph.
Return value:		True if right- and left hand side graphs are equal; false if they are not.

uUDGraph::Operator !=

Purpose:	Compares two graphs for logical inequality. Both the right- and left hand sides are const.	
Syntax:	1 `bool`	`uUDGraph& == uUDGraph&`
Parameters:	`uUDGraph&`	Reference to graph.
Return value:		True if right- and left hand side graphs are not equal; false if they are.

E.3 uUDGraphNode API

uUDGraphNode::ID

Purpose: Returns the serial number of the current graph node.

Syntax:
```
1    long    ID()
```

Parameters: None.

Return value: Serial number of the current uUDGraphNode object.

uUDGraphNode::Value

Purpose: Syntax 1 returns the name of the compartment; syntax 2 returns the current name and sets a new name. For syntax 2, the string reference parameter is const.

Syntax:
```
1    p3DPoint    Value()
```

Parameters: None.

Return value: Vector representing the coordinates of the current node.

NOTE: In order to prevent creation of multiple nodes having the same value, no value-setting function has been defined.

uUDGraphNode::begin

Purpose: Sets a list iterator to the first element of the graph node's connection list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

Syntax:
```
1    uUDGraphNodeItr         begin()
2    uUDGraphNodeConstItr    begin()
```

Parameters: None.

Return value: Iterator which refers to the first connection in the list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

uUDGraphNode::end

Purpose: Sets a list iterator to past the last element of the graph node's connection list. Syntax 1 returns a non-const iterator; syntax 2 returns a const iterator.

Syntax:
```
1    uUDGraphNodeItr         end()
2    uUDGraphNodeConstItr    end()
```

Parameters: None.

Return value:			Iterator which refers to one element past the last connection in the list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

<ins>uUDGraphNode::empty</ins>

Purpose: Returns state of the graph node's connection list.

Syntax:	1	`bool`	`empty()`
Parameters:			None.
Return value:			True if the graph node's connection list is empty; false if it is not.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

<ins>uUDGraphNode::size</ins>

Purpose: Returns the size of the graph node's connection list.

Syntax:	1	`int`	`size()`
Parameters:			None.
Return value:			Number of connections in the graph node's connection list.

NOTE: Lower case function name is used for consistency with STL nomenclature for container classes.

<ins>uUDGraphNode::Operator ==</ins>

Purpose: Compares values of two graph nodes for logical equality. Both the right- and left hand sides are const.

Syntax:	1	`bool`	`uUDGraphNode& == uUDGraphNode&`
Parameters:	`uUDGraphNode&`		Reference to graph node.
Return value:			True if values of right- and left hand side graph nodes are equal; false if they are not.

<ins>uUDGraphNode::Operator !=</ins>

Purpose: Compares values of two graph nodes for logical inequality. Both the right- and left hand sides are const.

Syntax:	1 `bool`	`uUDGraphNode& != uUDGraphNode&`
Parameters:	`uUDGraphNode&`	Reference to graph node.
Return value:		True if values of right- and left hand side graph nodes are not equal; false if they are.

<u>uUDGraphNode::Operator <<</u>

Purpose:	Inserts graph node into output stream. The graph node reference is const.	
Syntax:	1 `ostream&`	`ostream& << uUDGraphNode&`
Parameters:	`uUDGraphNode&`	Reference to a source graph node.
Return value:		Reference to an output stream.

<u>uUDGraphNode::Operator >></u>

Purpose:	Extracts graph node from input stream. The graph node reference is const.	
Syntax:	1 `istream&`	`istream& >> uUDGraphNode&`
Parameters:	`uUDGraphNode&`	Reference to a destination graph node.
Return value:		Reference to an input stream.