V encontro nacional de mecânica computacional
Universidade do Minho - Guimarães - 20 a 22 de Outubro de 1997

OBJECT ORIENTED PROGRAMMING: DATA PREPARATION AND VISUALIZATION OF *FEM* MODELS

Álvaro F.M. Azevedo[1],Joaquim A.O. Barros[2],Eduardo R.B. Marques[3] and Pedro S.O. Branco[3]

ABSTRACT

In this paper two object oriented applications are described. The former is intended to generate data associated with the finite element method (*FEM*) and the later is a three-dimensional visualization tool named *3DMesh*. Both are based on the principles of object oriented programming, namely encapsulation, inheritance and polymorphism. To support the preparation of *FEM* data, a language named *3DO* was developed. Its syntax is similar to a subset of the C++ programming language. *3DO* is based on object construction and modification by methods that require a small number of arguments. With this tool, mesh generation, definition of properties and loads and mesh refinement can be performed with limited user effort, even when the model is complex. All the generated information can be visualized with the program *3DMesh*. This application is based on the *OpenGL* library and uses the *Microsoft Foundation Classes* to simplify its integration in the *MS-Windows* environment. *3DMesh* implements an interactive navigation technique that allows the visualization of the model interior, preserving its integrity. Model attributes and the results of the *FEM* analysis can also be visualized.

1. INTRODUCTION

As computer performance increases, the finite element method becomes capable of analyzing models with higher complexity, in terms of shape, size, properties, loads and behavior. With this growing trend, the time dedicated to the data preparation phase becomes unacceptably long and new ways of generating huge amounts of information must be

1 - Assistant Prof., Faculty of Engineering, University of Porto, 4099 Porto Codex
2 - Assistant Prof., University of Minho, Civil Engineering Department, Azurém, 4800 Guimarães
3 - Degree in Computer Science, Science Faculty, University of Porto

envisaged. Furthermore, when changes have to be introduced in existing models, the redefinition of all the properties and loads may become a daunting task. When this is necessary, traditional methods such as element enumeration or *CAD* require a tedious repetition of most of the data preparation work. With an object oriented approach and with the help of a dedicated language, *FEM* data generation becomes more effective.

When 3D models are complex or hollow, and their interior parts have to be observed, traditional strategies are based on model slicing or model decomposition, followed by the removal of the parts that hide its interior. With the program *3DMesh* no slicing or decomposition is required, since an interactive walk-through navigation is implemented. During this visit to the interior of the model, its attributes, such as material properties, loads, stresses, etc. can be observed.

2. DATA PREPARATION USING THE *3DO* LANGUAGE

3DO is a high level language dedicated to the generation of three-dimensional geometric entities and their physical attributes, in accordance with finite element techniques [Zienkiewicz and Taylor 1989]. The *3DO* interpreter parses the statements, that must be previously prepared using a text editor, and generates a new file containing a comprehensive description of the *FEM* model.

3DO is based on two concepts: objects and operations on objects. These objects can represent several types of entities, being the finite element the most important. Apart from three-dimensional geometrical characteristics, a finite element has physical attributes, such as material properties and loads. Object properties and attributes are created with default values that can subsequently be changed using *3DO* operations.

A simple example is used to introduce the main features of the *3DO* language (Fig. 1). The complete description of the language can be found in Marques *et al*. 1997.

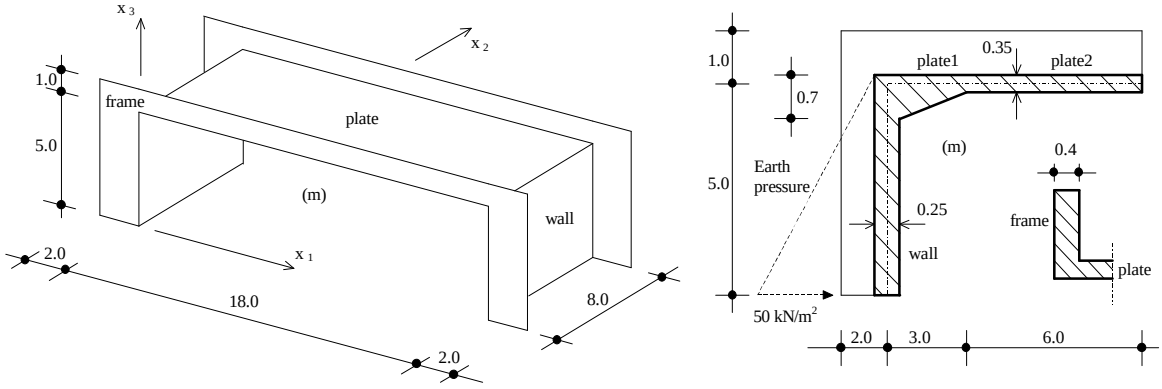
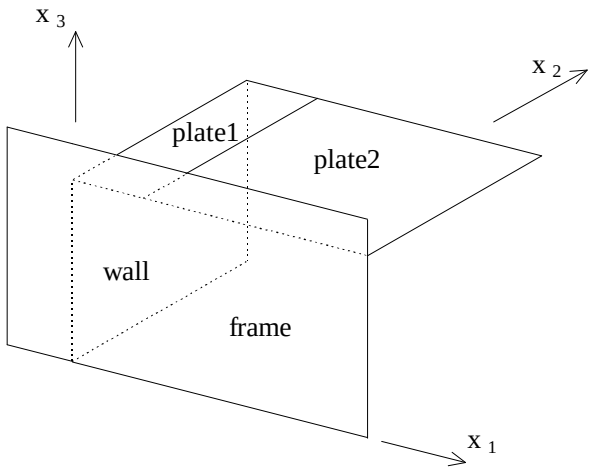


Fig. 1 - Bridge geometry, thickness of structural components and loads to be modeled using the *3DO* language.

The bridge's main geometric components are created by the *3DO* code fragment listed in Fig. 2a. Instruction (1) creates a *Layer* object named *Bridge*. When the layer is selected (see instruction (2)) all the objects subsequently created are stored in that layer. Multiple layer objects might be used in order to decompose the structure in sub-structures, allowing for an independent manipulation of each layer or set of layers. The structure is discretized by four noded Lagrange elements named *Quad4* (see instructions (9), (12), (15) and (21)). Each *Quad4* is defined by four previously created *Point* objects (instructions (5-8), (10), (11), (13), (14) and (17-20)). Auxiliary objects of type *Double* are used to specify shared coordinates to

be used in the construction of *Point* objects. This technique allows a parametric definition of the model components (instructions (3) and (4)). Auxiliary objects are not stored in the output file.

```
(1) Layer Bridge("Demonstration example");
(2) SetCurrentLayer(Bridge);

// Define wall
(3) Double Width 8.0; // wall width = 8.0
(4) Double Height 5.0; // wall height = 5.0

(5) Point wall_p1(0.0, 0.0, 0.0);
(6) Point wall_p2(0.0, 0.0, Height);
(7) Point wall_p3(0.0, Width, Height);
(8) Point wall_p4(0.0, Width, 0.0);

(9) Quad4 wall(wall_p1,wall_p2,wall_p3,wall_p4);

// Define plate1 and plate2
(10) Point plate1_p2(3.0, 0.0, Height);
(11) Point plate1_p3(3.0, Width, Height);

(12) Quad4 plate1(wall_p2, plate1_p2,
                  plate1_p3, wall_p3);

(13) Point plate2_p2(9.0, 0.0, Height);
(14) Point plate2_p3(9.0, Width, Height);

(15) Quad4 plate2(plate1_p2,plate2_p2,
                  plate2_p3,plate1_p3);

// Define frame
(16) Double Frame_Height 6.0;

(17) Point frame_p1(-2.0, 0.0, 0.0);
(18) Point frame_p2(9.0, 0.0, 0.0);
(19) Point frame_p3(9.0, 0.0,  Frame_Height);
(20) Point frame_p4(-2.0, 0.0,  Frame_Height);

(21) Quad4 frame(frame_p1,frame_p2,
                 frame_p3,frame_p4);
```



a) *3DO* code b) Model visualization

Fig. 2 - First step of the model generation using the *3DO* language. Construction of the primary elements.

3DO attributes can be scalar values, vectors or text labels. These attributes can be attached to finite element objects or to *Point* objects. Element attributes can simulate discontinuities between elements, such as thickness or material properties. Two element attributes are defined in the *3DO* code listed in Fig. 3. The first attribute (instruction (1)) represents the thickness of the elements shown in Fig. 2b. The thickness of *plate1* varies linearly from 0.70 m to 0.35 m (see Fig. 1) and is defined by instruction (2). The thickness of the remaining elements is constant: *plate2*, *wall* and *frame* are 0.35 m, 0.25 m and 0.40 m thick, respectively (instructions (3-5)). The earth pressure on the bridge walls is characterized by the attribute defined in instructions (6) and (7).

```
// Define element thickness and earth pressure attributes

(1) ElementScalarField thickness("Element  thickness", 0.0);

(2) plate1.ElementScalarValues(thickness, 0.7, 0.35, 0.35, 0.7);
(3) plate2.ElementScalarValues(thickness, 0.35);
(4) wall.ElementScalarValues(thickness, 0.25);
(5) frame.ElementScalarValues(thickness,0.4);

(6) ElementVectField  earth_pressure ("Wall: earth pressure",(0.0, 0.0, 0.0));

(7) wall.ElementVectValues( earth_pressure, (50.0, 0.0, 0.0), (0.0, 0.0, 0.0),
                                            (0.0, 0.0, 0.0), (50.0, 0.0, 0.0));
```

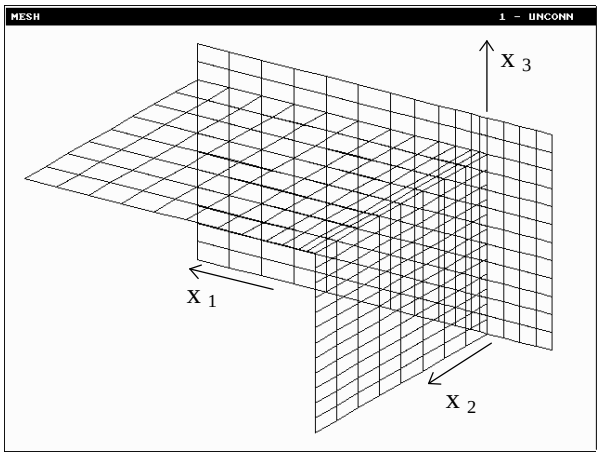
Fig. 3 - Second step of the model generation. Attribute definition and assignment.

The *Refine* operation divides a finite element in smaller elements. The nodal coordinates of the generated elements are calculated using the element shape functions [Hinton and Owen 1979]. Physical attributes are inherited by the generated elements and their values are interpolated using the same shape functions.

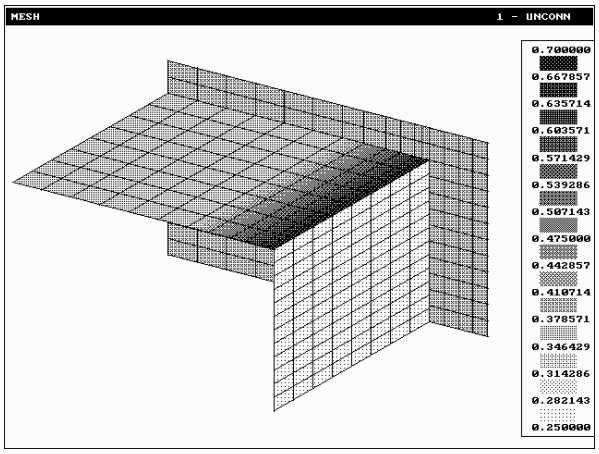
The *3DO* code that performs the mesh refinement is listed in Fig. 4a. Several objects of type *Weights* (instructions (1-3), (6) and (7)) must be supplied as parameters of the *Refine* operation (instructions (4), (5), (8) and (9)). Each object of type *Weights* defines the number of divisions and their proportions along one element edge. Fig. 4b shows the refined mesh and Fig.s 4c and 4d show the thickness and earth pressure attributes after mesh refinement.

```
// Refine the primary elements
(1) Weights w_plate1_s1(6, 0.25, 0.25, 0.50,
                           0.50, 0.75, 0.75);
(2) Weights w_plate2_s1(6);

(3) Weights w_wall_plates_s2(8);

(4) plate1.Refine(w_plate1_s1,w_wall_plates_s2);
(5) plate2.Refine(w_plate2_s1,w_wall_plates_s2);

(6) Weights w_frame_s1(16,
                       0.50, 0.50, 0.50, 0.50,
                       0.25, 0.25, 0.50, 0.50,
                       0.75, 0.75, 1.00, 1.00,
                       1.00, 1.00, 1.00, 1.00);

(7) Weights w_frame_s2(12);

(8) wall.Refine(w_frame_s2,w_wall_plates_s2);
(9) frame.Refine(w_frame_s1,w_frame_s2);
```

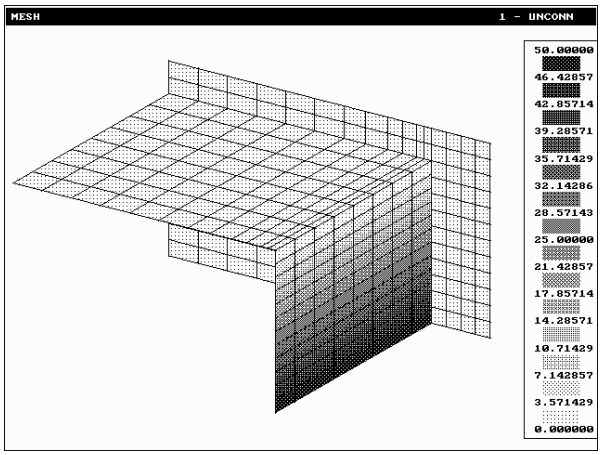
a) *3DO* code



b) Refined mesh



c) Thickness attribute (m)



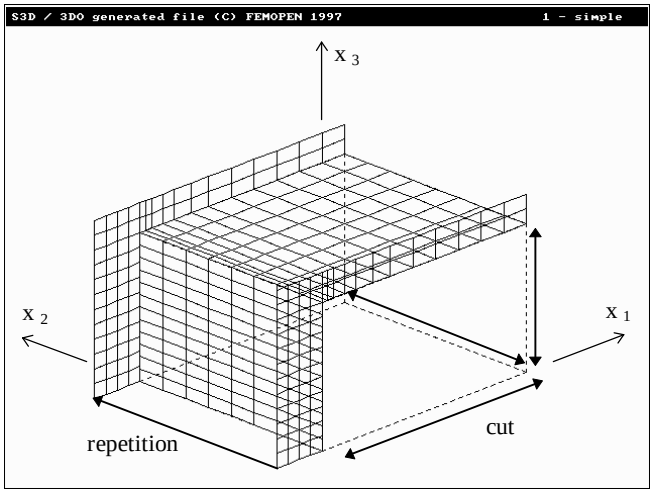
d) Earth pressure attribute (kN/m^2)

Fig. 4 - Third step of the model generation. Refinement procedure.

The first instruction of the code listed in Fig. 5a repeats the *frame* object along the x_2 axis. The physical attributes of the *frame* are also copied to the newly created object. Subsequent operations on the *frame* object affect both copies. In instruction (2), a *PolygonalRegion* auxiliary object named *frame_cut* is defined. A *PolygonalRegion* is a prism defined by a set of points contained in a plane and by the prism height. In this example, four points and a thickness define the *frame_cut* object (see Fig. 5a). Objects inside regions can be grouped using the *RegionSelection* operator (instruction (3)). The properties of the objects in

the group can be globally changed using object operations. In instruction (3) the *Freeze* operation hides the elements of the object *frame* that lie within the *frame_cut* region.

```
// Create a frame copy
(1) frame.Repeat((0.0, Width, 0.0),2);

// Remove some frame elements
(2) PolygonalRegion  frame_cut(4,
                          (-0.1, 0.0, -0.1),
                          ( 9.1, 0.0, -0.1),
                          ( 9.1, 0.0,  5.1),
                          (-0.1, 0.0,  5.1),
                            8.1);

(3) frame.RegionSelection( frame_cut).Freeze;
```



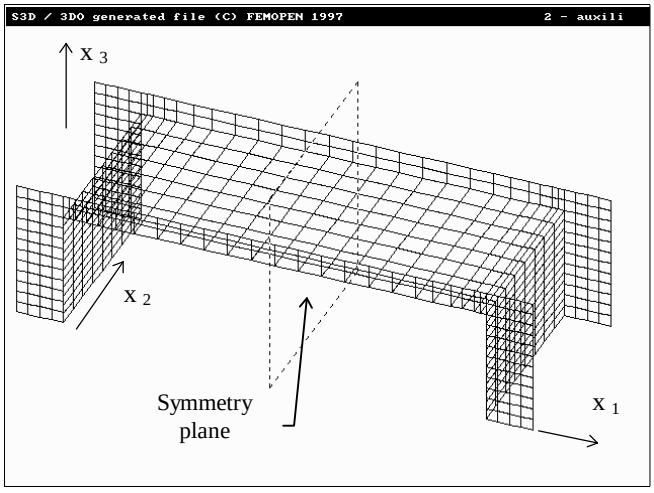
a) *3DO* code b) Visualization

Fig. 5 - Fourth step of the model generation. Definition of the frame structure.

One half of the bridge is already defined. The second part is created by mirroring a copy of the first half. The corresponding code is listed in Fig. 6a. The symmetry plane is characterized by a *Plane* object named *plane_x9* (instruction (1)).

An object of type *Collection* groups copies of previously existing objects. In instruction (2), copies of *plate1*, *plate2*, *wall* and *frame* are created and grouped in the *Collection* object named *coll*. In instruction (3) *coll* is mirrored.

```
// Make copies of plate1, plate2, wall and
// frame, grouping them in a Collection
// object.
// The grouped objects are mirrored
// through plane x1=9

(1) Plane plane_x9(yz,9.0);

(2) Collection coll(wall,plate1,plate2,frame);

(3) coll.Mirror(plane_x9);
```



a) *3DO* code b) Visualization

Fig. 6 - Fifth step of the model generation. *Mirror* operation.

Tables 1 and 2 briefly describe the most relevant *3DO* objects and operations.

Table 1 - *3DO* objects.

Type	**Name**	**Construction arguments**	**Comments**
Auxiliary	*Double*	Floating point constant.	-
	Coords	3 *Double* objects.	3D point coordinates.
	Axis	2 *Coords* objects.	-
	Plane	3 *Coords* objects or a macro (*xy*, *xz* or *yz*) and a *Double* object.	-
	Weights	Integer constant *n* and *n* *Double* objects.	Number of divisions and weighting factors along one edge of a finite element.
	PolygonalRegion	Integer constant *n*, *n* *Coords* objects and a *Double* object.	Number of vertexes of the polygonal region, coordinates of the vertexes and thickness.
Point	*Point*	3 *Double* objects.	-
Finite elements	*Line2*; *Line3*	2 or 3 *Point* objects.	One-dimensional finite element with 2 or 3 nodes.
	Triang3	3 *Point* objects.	Two-dimensional finite element with 3 nodes.
	Quad4; *Quad8*; *Quad9*	4, 8 or 9 *Point* objects.	Two-dimensional finite element with 4, 8 or 9 nodes.
	Brick8; *Brick20*	8 or 20 *Point* objects.	Three-dimensional finite element with 8 or 20 nodes.
Containers	*Collection*	Objects to be grouped.	Auxiliary group.
	Layer	Layer title.	Model subsection.
Attributes	*PointScalarField* *PointVectField* *PointLabelField*	Title and default value.	Attributes of the points.
	ElementScalarField *ElementVectField* *ElementLabelField*	Title and default value.	Attributes of the elements.

Table 2 - *3DO* operations.

Name	**Target** (1)	**Arguments**	**Comments**
PointScalarValues *PointVectValues* *PointLabelValues*	Points	Point attribute identifier and a scalar, vector or label variable.	Definition of point attributes.
ElementScalarValues *ElementVectValues* *ElementLabelValues*	Finite elements	Element attribute identifier and a list of scalars, vectors or labels.	Definition of element attributes.
Refine	Finite elements	List of *Weights*.	Refines the finite element.
RegionSelection	Containers	*PolygonalRegion* object.	All objects inside the region are selected.
Freeze	Any object	None.	Hides the object.
Mirror	Any object	*Plane* object.	Mirrors the object through the plane.
Move	Any object	3 *Double* objects.	Each double specifies the displacement component along the corresponding axis.
Repeat	Any object	3 *Double* objects and an integer constant *n*.	Repeats the object *n* times in the specified direction.
Rotate	Any object	*Axis* object *A* and *Double* object *D*.	Rotates the object *D* degrees using *A* as the rotation axis.
Scale	Any object	*Double* object *F*.	Reduces or enlarges the object according to the factor *F*.
Copy	None	Any existent object and identifier of the object to be created.	Creates a copy of an existing object.

(1) Operations cannot be performed on auxiliary objects or attributes.

The advantages of the *3DO* language become more evident in complex mesh generation problems, such as the one shown in Fig. 7.

Fig. 7 - Complex three-dimensional model.

3. INTERACTIVE NAVIGATION IN 3D MODELS

Scientific data interpretation is a hard task without the support of a suitable graphical tool. When 3D models are too complex (see Fig. 7), a global visualization is inconvenient. In these cases the model has to be sliced or decomposed to allow the observation of the hidden parts [Gross 1994]. The *3DMesh* program, described in this section, implements an interactive navigation environment in order to achieve a perception of the model interior, preserving its integrity. *3DMesh* can be used to visualize the models generated with the *3DO* language (see Section 2). By means of a specific neutral file, the results of a scientific analysis can be visualized as vectors or contour fills.

3DMesh is an object oriented program written in C++ that uses the following libraries: *Microsoft Foundation Classes* (*MFC*) [*Microsoft* 1995] to provide a standard *MS-Windows* interface and *OpenGL* [Neider 1993] for efficient graphical output. As the combination of these libraries is not straightforward, a dedicated interface is required [Fosner 1997].

3.1 Class hierarchy

The classes described in Section 2 are not reused in *3DMesh*, due to the specific characteristics of each program. Fig. 8 shows the class hierarchy used in *3DMesh*.

The following basic classes were defined: *CNode* encapsulates the nodal coordinates and *CVector*, derived from *CNode*, has an additional method for vector normalization.

The *CFiniteElement* class defines the characteristics that are shared by all its derived classes. Each finite element type has its own class derived from *CFiniteElement* and contains an array of *CNode* and an array of *CVector*. The former represents the nodal coordinates and the later the nodal normal vectors required by the *OpenGL* library.

Each finite element object has methods for drawing itself, writing its labels, painting its scalar field and evaluating the shape functions that are necessary to calculate the normal vector in each node [Zienkiewicz and Taylor 1988].

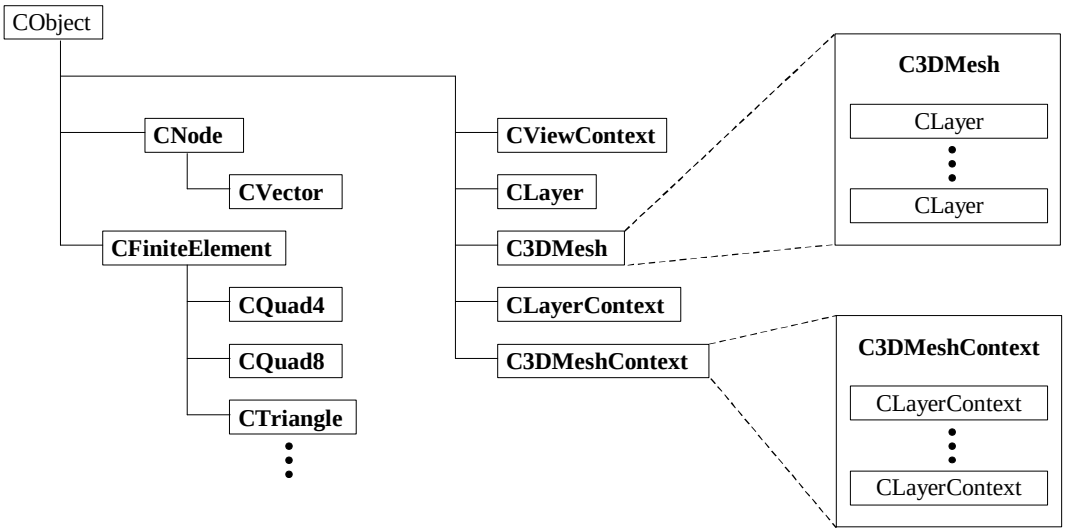


Fig. 8 - *3DMesh* class hierarchy.

A layer is a set of elements that share the same properties, such as color, texture and visibility. The class *CLayer* encapsulates the functionality of a layer. Its draw method calls the draw method of each finite element. All the properties are aggregated in a distinct class called *CLayerContext*, so that a layer can be simultaneously displayed with different properties in different windows. The class *C3DMesh* aggregates all the layers that are present in the input file. Similarly, *C3DMeshContext* contains an array of *CLayerContext*, with a number of elements that coincides with the number of layers (see Fig. 8).

The graphical output is simplified by the aforementioned aggregation, since all the model components are drawn with a single call to the *C3DMesh* draw method.

The *CViewContext* class encapsulates the user viewpoint coordinates and the model orientation in the scene. The viewpoint or the model can be translated or rotated by *CViewContext* methods.

3.2 *MFC-OpenGL* integration

The integration of a program with the *MS-Windows* environment becomes easier when an appropriate library of classes and functions is employed. *3DMesh* uses the *Microsoft Foundation Classes* (*MFC*) and their associated document/view framework. *MFC* encapsulates the functionality of several interface components, such as menus, dialog boxes, toolbars, etc. In order to benefit from the object oriented programming abstraction paradigm, all the user developed modules must be object oriented.

OpenGL is a subroutine library whose main purpose is to render 3D models, with lighting and optional textures. Its availability on most platforms and its integration with dedicated graphics hardware make *OpenGL* an efficient and standard tool that helps the development of rendering software. The *OpenGL* library contains functions that perform translations, rotations, viewport manipulation, clipping, lighting, texture mapping, etc.

Since *OpenGL*'s design is not object oriented, its integration with the *MFC* framework requires a suitable interface class. In *3DMesh* a class named *COpenGLView* [Fosner 1997] is used to initialize the *OpenGL* environment, whenever a new view is created. *COpenGLView* is responsible for setting up the *OpenGL* rendering context, as well as resizing the viewport when the window size changes.

Four additional classes are derived from *COpenGLView*: *CMainView*, *CTopView*, *CFrontView* and *CSideView*. Their role is to provide four different interfaces between the user and the data model (see Fig. 9). Several *CMainView* instances may be created, allowing

multiple views of the model with different drawing contexts. Only *CMainView* accepts user input by means of keystroke combinations that change the viewpoint position or, otherwise, translate or rotate the object. The viewpoint can be interactively moved to the model interior (see Fig. 10). Only the model components that fall inside the clipping volume (frustum) [Wright and Sweet 1996] are represented. *CTopView*, *CFrontView*, and *CSideView* are auxiliary views, whose role is to show the wireframe representation of the three basic orthogonal projections of the model (see Fig. 10).

C3DMeshDoc is derived from the *MFC* class *CDocument* and contains a *C3DMesh* object describing all the model data.

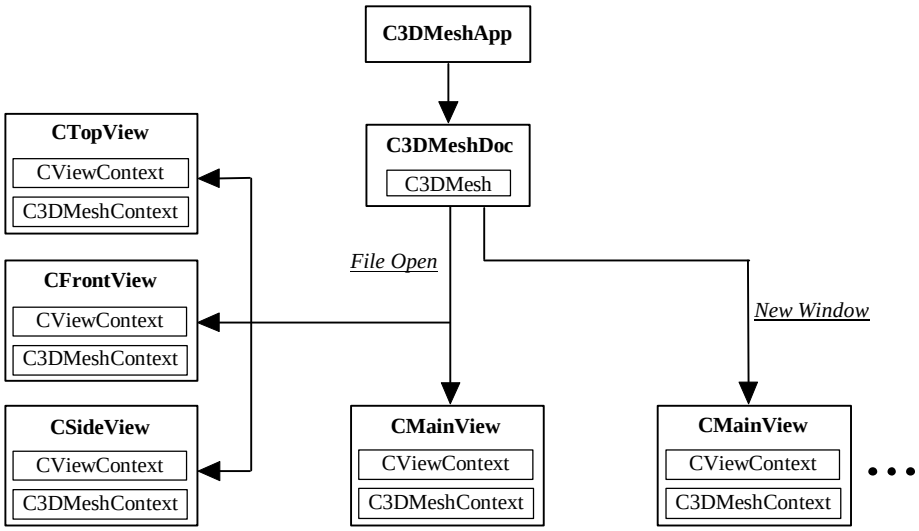


Fig. 9 - *3DMesh*: document/view architecture - window creation diagram.

A screen shot of the graphical user interface of the program *3DMesh* is shown in Fig. 10. Three orthogonal views of a gallery and a perspective from an interior viewpoint are displayed in child windows. Several dialog boxes and toolbar buttons are available to allow the modification of layer properties, such as colors, visibility, line types, fonts, etc. The model can be rendered in solid mode, in wireframe mode or with an overlapped contour fill representing a scalar field.

4. CONCLUSIONS

The advantages of data preparation with the *3DO* language became evident in these early tests and encourage further developments, mainly in the attribute generation and manipulation. Since the program output is a comprehensive and easily interpretable file, the generated models can be used in many scientific domains.

3DMesh is a practical and versatile visualization tool, well integrated in modern computational platforms and fully exploiting the performance of dedicated *OpenGL* hardware. Since its input is a neutral file, *3DMesh* can be used to visualize information proceeding from numerous kinds of problems.

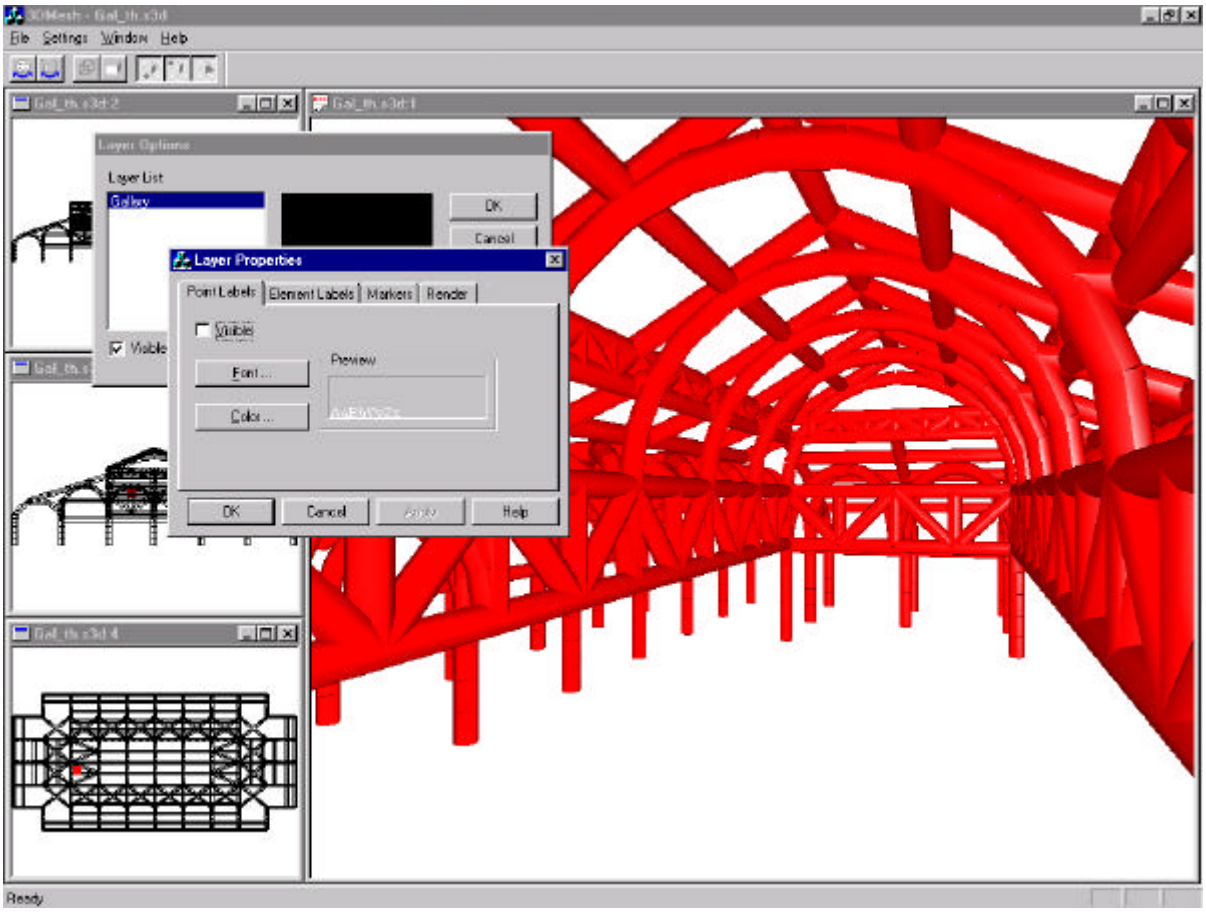


Fig. 10 - Gallery: three orthogonal projections and inner view.

REFERENCES

Fosner, R. (1997), *OpenGL Programming for Windows 95 and Windows NT*, Addison-Wesley.

Gross, M.H. (1994), *Subspace Methods for the Visualization of Multidimensional Data Sets*, in Scientific Visualization - Advances and Challenges, Edited by Rosenblum *et al*., Academic Press, pp. 171-186.

Hinton, E. and Owen, D.R.J. (1979), *An Introduction to Finite Element Computations*, Pineridge Press, Swansea, U.K.

Marques, E.R.B., Azevedo, A.F.M. and Barros, J.A.O. (1997), *3DO Users Manual*, FEMopen Software, Lda., Porto, Portugal.

Microsoft Foundation Class Library Reference (1995), Microsoft Press.

Neider, J., Davis, T. and Woo, M. (1993), *OpenGL Programming Guide*, Addison-Wesley.

Wright, R. and Sweet, M. (1996), *OpenGL Superbible*, Waite Group Press.

Zienkiewicz, O.C. and Taylor, R.L. (1988), *The Finite Element Method*, 4th Edition, McGraw-Hill.