ISSN 1392 – 124X INFORMACINĖS TECHNOLOGIJOS IR VALDYMAS, 2003, Nr.4(29)

OBJECT-ORIENTED IMPLEMENTATION OF TRANSIENT WAVE PROPAGATION FINITE ELEMENT MODELS

Rimantas Barauskas, Mindaugas Kuprys, Ula Leonavičiūtė

Kaunas University of Technology
Studentų str. 50-407, LT-3031 Kaunas

Abstract. The possibilities to use the object-oriented programming methodology for transient elastic wave propagation finite element models are discussed. A comparative analysis of functional, procedural and object-oriented programming approaches for the finite element method is presented. The object-oriented method is demonstrated to have significant advantages for development of the finite element transient dynamics application.

1. Introduction

The object-oriented programming (OOP) is widely accepted as a methodology for writing modular and reusable code. Popularity of the OOP as an alternative to traditional, procedural or functional programming methods is growing fast as it provides significant improvements that allow construction and maintenance of the complex software systems.

The applications of the finite element analysis require one of the highest codes writing standards in order to ensure the effective and quick program execution, as well as, reusing and sharing of the code for the further development of the program. These features demand the change from the steady procedural programming to the higher quality programming. The OOP seems to be well suited for such needs [7].

In recent years a lot of work has been done in rewriting traditional FE procedures into object-oriented (OO) code without redesigning the whole program. In this way higher functionality cannot be guaranteed, however it illustrates perfectly the capabilities of OO approach. In reality, production of the efficient OO programs involves much more than a simple translation of FORTRAN, C or any other procedural language codes to any OO language [7].

Why OO technique is better to use in FE model? There is no unambiguous answer to this question. First of all it strictly depends upon the nature of the problem. If the pending task is simple, the OO isn't the best solution. On the other hand, if the problem is complex, the set of arguments can be easily declared which will lead to OO approach appliance. An increasing number of OO packages have been already developed by different authors to solve wide range of problems. Each of them provides different technique dependent on the problem. The field of finite element analysis where OPP has been used is quite wide, starting from stress analysis [4], structural dynamics [10], shell structures [9], non-linear plastic strain [8] and ending with the contact problems [11]. The application used as an example in this paper is transient wave propagation in an elastic media.

The aim of this paper is to explain and illustrate the advantages of OOP in the application area of transient dynamics problems with particular emphasis to simulations of the wave propagation in an elastic environment. A description of implemented classes of their hierarchy and some details of implementation are also provided in this paper.

The next section will explain some of the basic programming techniques, as well as, the main OO features for the finite element method (FEM) will be discussed. The third section will explain the basic steps of the finite element (FE) program. Section 4 will explain OO implementation of the program. A hierarchy of classes is proposed and discussed; it provides a description of all main classes and realizations of the methods. The last section will illustrate the results obtained by the program.

2. Programming approaches for the FEM application

2.1. Procedural, Functional and Object- Oriented programming

The three major programming approaches used for FEM implementations are:

- Object-oriented programming (C++, C#, JAVA, etc.);
- Procedural programming (C, Visual Basic, FORTRAN, Ada, etc.);

- Functional programming (Lisp, MatLab, Mathematica, ANSYS, etc.);

Functional programming languages are close to procedural ones. They provide many built-in functions, easy operations with data and visualization tools. In the FE case, functional languages are useful for solving some non-complex tasks, e.g., static's problems. Computation time is significantly longer if dynamic tasks such as heat transfer, transient wave propagation or contact problems are being solved.

Some programming languages combine two or three programming approaches. For instance, C++ is procedural, functional, and object-oriented, while Visual Basic is procedural and functional [5]. Advantages of using debugging and development tools such as Microsoft Visual Studio or Borland C++ Builder, together with graphics library OpenGL, enables the development of the highest quality FE applications for almost any engineering problem. The program used as an example in this paper is developed with Visual Studio 6.0 by integrating OpenGL for graphical visualization. Another advantage of compiled languages like C++ is that the executable program can be distributed to people who don't have the compiler. Mathematically oriented programming environments like MatLab, are more interpreter languages; interpreters are necessary to run the code.

2.2. The main OO features for the FE application

OO programming technique is widely used to develop many complex scientific programs; FEM programs are not an exception. In recent years OOP has proved to be one of the easiest, fastest and most efficient ways to develop FE applications.

There are three basic OO properties such as *encapsulation (data hiding), inheritance and polymorphism* that are combined together in order to give OOP features which are useful for FE programs. The basic components of the FEM, like the node, the element, the material, can easily be fitted into an objects world, with their own behavior, state and identity [2].

Encapsulation and data hiding

Encapsulation and data hiding is the first and the most important idea in the OOP design. Data hiding is a highly valued feature which enables the use of an object without even knowing how it works internally. Mostly all of the attributes and methods are *private*, which means they cannot be seen or altered by any other object. Data exchange with other methods is being performed only through *public* methods [3]. Public functions known as accessor methods should be created in order to set or get the *private* member variables. The property of being self-contained unit is called *encapsulation* [6]. The main advantage of data hiding is the safe use of class variables. "Safe use" means that variables are accessed only by class methods and cannot be changed by any other classes that are not permitted for them.

In the FE application developed by any OO language, encapsulation (data hiding) is used almost everywhere. For instance, in the program discussed in this paper (wave propagation) the class *DynamicEquation* does not have value "number of elements" (*m_numberOfElements*) in the structure. The value *m_numberOfElements* is a *private* member of class *General*. If we need to get this value, first of all we have to initialize the pointer (**g_pointer*) to *General* in class *DynamicEquation* and to access it by using public accessor *Get_NumberOfElements:*

```
//---
    int numOfElem = 0; // number of elements
    General *g_pointer; // pointer to General
    numOfElem=g_pointer->
       Get_NumberOfElements();
//---
```

Inheritance

Inheritance is the process by which a class inherits the properties of its base class. Methods and instance variables are inherited. As an OO language C++ supports the idea of reuse through inheritance. New subclass is derived from its base class. If we have derived a new class, we can easily create new objects.

In the FE case the inheritance can be used in several ways. Assume that the analyzed structure is subdivided into a number of finite elements consisting of three or four nodes. In our example, triangle finite elements have been used; however it can be easily extended for quadrilateral and other element shapes. Inheritance, here, could be found as the best solution. An abstraction of an element helps to derive new classes describing the finite element. So, it is point to have some general class *Element* which would be parent-class for all derived subclasses (e.g., *TriangleElement*). It could have such attributes as element identifier, material identifier and element thickness. All derived classes could have their own attributes. In C++ it looks like:

```
//---
class Element
{
private:
    int m_elemNr; // element number
    int m_mat_num; // material number
    double m_h; // element thickness
public:
    . . .   // methods
};
class TriangleElement : public Element
{
private:
    int m_firstNode;    // first node
    int m_secondNode;   // second node
    int m_thirdNode;    // third node
    double m_width; // element width
    double m_S; // square of an element
public:
    . . .   // methods
};
//---
```

The presented piece of code demonstrates the definition of the abstract class *Element* with its private

members. There are three private data members defined in this class: *m_elemNr* (number of an element), *m_mat_num* (number of a particular element material) and *m_h* (thickness of an element). These variables do not depend upon the shape of the finite element. Other variables *m_width* (width of an element), *m_S* (area of an element) and nodes numbers should be defined in a particular class, currently in the *Triangle-Element* class.

More details about the structure of the classes' hierarchy presented in section 4.1.

Polymorphism

Polymorphism can be considered as a unique feature of OOP allowing different objects to respond to the same message in their own specific way. In our example polymorphism feature wasn't used, however it certainly will be used in the future for extensibility of the application.

3. Basic steps of the FE program

3.1. Basic equations of the transient dynamic analysis

Transient wave propagation analysis is performed by solving the structural dynamic equation:

$$[\mathbf{M}]\{\ddot{\mathbf{U}}\}+[\mathbf{K}]\{\mathbf{U}\}=\{\mathbf{F}(\mathbf{t})\}, \qquad (1)$$

where $[\mathbf{M}]$, $[\mathbf{K}]$ – the structure mass and stiffness matrices, respectively; $\{\mathbf{F}\}$ – the external load vector; $\{\mathbf{U}\}$, $\{\ddot{\mathbf{U}}\}$ are the nodal displacement and acceleration vectors of the structure.

The stiffness matrix $[\mathbf{K}]$ of the complete structure is obtained by assembling together the stiffness matrices $[\mathbf{K}^e]=\mathbf{h}\int\limits_{V}\int\int[\mathbf{B}]^T[\mathbf{D}][\mathbf{B}]\mathbf{dV}$ of each individual element, where $[\mathbf{B}]$ – matrix defining the relation between the strain vector and the nodal displacements, $[\mathbf{D}]$ – media stiffness tensor.

The matrix $[\mathbf{M}^e]=\frac{\rho h S_\Delta}{3}\begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\0&0&1&0&0&0\\0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\end{bmatrix}$ is the lumped (diagonal) mass matrix, where ρ – density, h – thickness of the element; S_Δ – area of the triangle. The matrix $[\mathbf{M}^e]$ is obtained by lumping the mass in equal parts between the nodes of the element [1]. The assembly of the elements matrices together results in the structural matrices $[\mathbf{K}]$ and $[\mathbf{M}]$.

Full investigation of the wave process dynamics can be performed by integrating the dynamic equation numerically. The Central Difference time integration scheme reads as

$$\{\mathbf{U}_{t+\Delta t}\}=[\hat{\mathbf{M}}]^{-1}\left[\begin{matrix}\{\mathbf{F}_t\}-\left([\mathbf{K}]-\frac{2}{\Delta t^2}[\mathbf{M}]\right)\{\mathbf{U}_t\}-\\ [\tilde{\mathbf{M}}]\{\mathbf{U}_{t-\Delta t}\}\end{matrix}\right], \qquad (2)$$

where $[\hat{\mathbf{M}}]=[\tilde{\mathbf{M}}]=\frac{1}{\Delta t^2}[\mathbf{M}]$ (3), Δt is the time integration step. The initial conditions are assumed to be $\{\mathbf{U}\}_0=0$, $\{\dot{\mathbf{U}}\}_0=0$;

3.2. Time step selection

The Central Difference Scheme (CDS) is said stable if the upper bound of time integration step Δt is:

$$\Delta t \le \frac{l_{\min}}{c}, \qquad (4)$$

where $l_{\min}$ – minimal linear dimension of the smallest finite element in the structure.

3.3. Post processing

The post processing step involves calculations of stresses in elements by using obtained values of nodal displacements as:

$$\{\sigma\}=\begin{Bmatrix}\sigma_x\\ \sigma_y\\ \tau_{xy}\end{Bmatrix}=[\mathbf{D}]\cdot[\mathbf{B}]\cdot[\mathbf{U}^e]=[\mathbf{G}]\cdot[\mathbf{U}^e], \qquad (5)$$

where $[\mathbf{G}]$ – stress matrix of the element, σ_x, σ_y – longitudinal stresses and τ_{xy} – shear stresses.

Two different external loading schemes were analyzed in the computed examples. The first research was performed by applying the step load. The excitation vector gets its value at time point t=0 and remains $\{\mathbf{F}(\mathbf{t})\}=const$ over all the simulation time T.

The second investigation takes the load vector as

$$F(t)=A\cdot\sin(\omega\cdot t), \qquad (6)$$

where t – pending time step, A – excitation amplitude; $\omega=\frac{2\pi}{T}$ – angular frequency.

The equivalent stresses in the i-th element are calculated as

$$\sigma_i=\sqrt{\frac{1}{2}(\sigma_{ix}-\sigma_{iy})^2+\frac{1}{2}\sigma_{ix}^2+\frac{1}{2}\sigma_{iy}^2+3\tau_{ixy}^2}. \qquad (7)$$

3.4. Overall flowchart

The flowchart diagram presented in Figure 1, illustrates the general algorithm for solving the transient dynamic problems in the FE method. The comments in rectangular callouts describe the particular case shown in this paper.

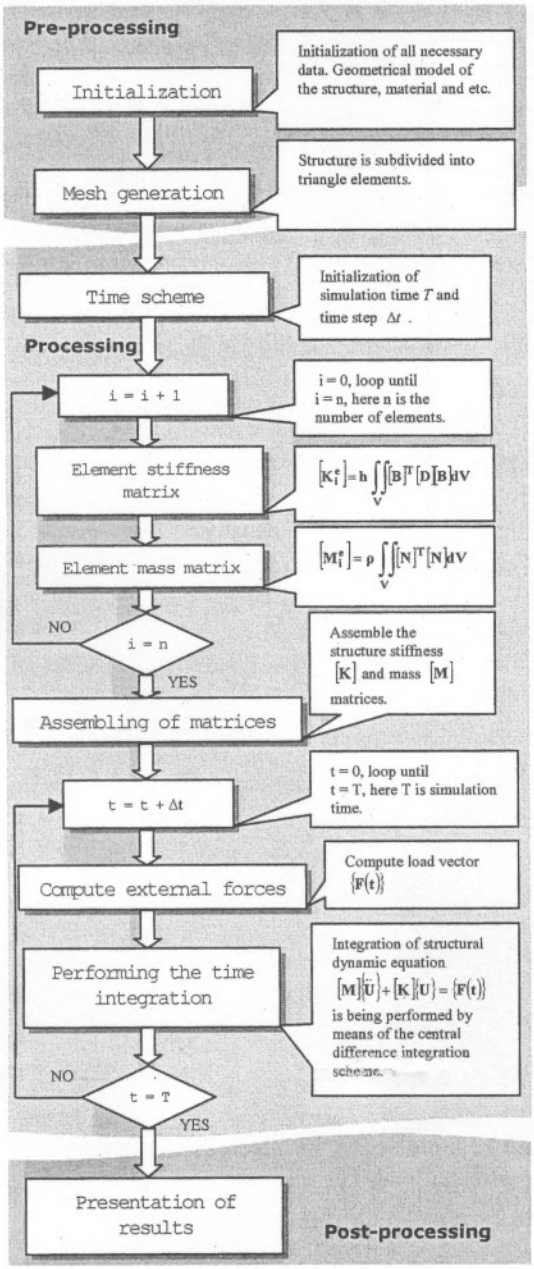

Figure 1. General algorithm of the FEM

The pre-processing stage involves the preparation of the data for calculations. Here the geometrical model of the structure is composed and meshed by subdividing it into a number of triangle elements. Material properties, surface loads and initial nodal displacements are assigned.

The processing stage involves all computational operations. The main task in this stage is to solve the dynamic equation (1). First of all, time settings should be set (step "Time scheme"). Next the structural stiffness ($[\mathbf{K}]$) and mass ($[\mathbf{M}]$) matrices have to be assembled, and the external load vector ($\{\mathbf{F}\}$) has to be computed. Then, the central Difference Scheme (CDS) is used to integrate numerically the equation.

The post-processing stage is designed for the presentation of the results. At each time step, the propagating wave is represented by the equivalent stresses (7) contour plot. Results are visualized in the 3D view by using Visual C++ and OpenGL tools.

4. OO implementation of the program

4.1. Class hierarchy

The class hierarchy of the code is reviewed and a brief description of each class is given in this section (see Figure 2). Subclasses are indicated under their parent-classes with a right indent.

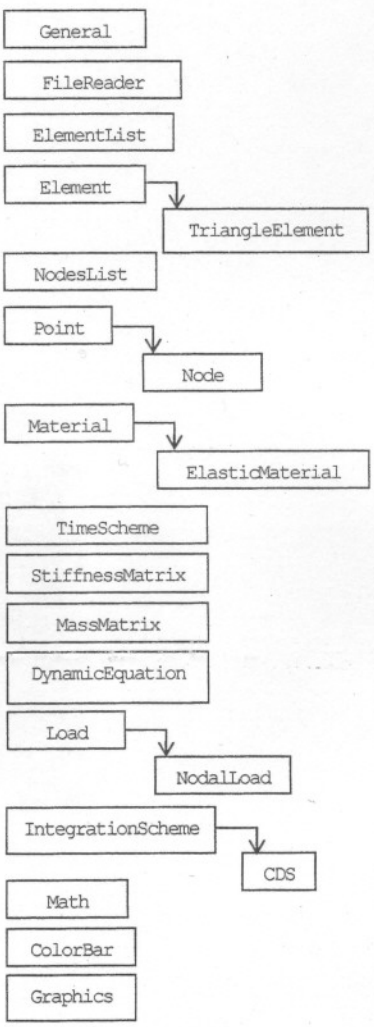

Figure 2. The class hierarchy

The design of class structure is developed according to two primary conditions: the particular class

should implement particular task and if it is possible, it should be reusable in other program.

Class General

The General class is the main one in all class hierarchy. It manages the whole solution algorithm. The class contains the data describing the problem size such as: number of nodes, number of elements, number of nodes of an element, etc. The principal tasks of this class are:

- receive the messages from the user and initiate the corresponding operations;
- manage all necessary operations to solve the problem.

Class FileReader

The FileReader class reads all the data from the data file and writes it to some kind of list. The principal tasks performed by the class are:

- create elements list;
- create nodes list;
- create materials list.

Class ElementList

The class manages the list of the structure elements. The class principal tasks are:

- to manage elements list;
- to add, delete or return an element of the list.

Class Element

This class is an abstract class for all kinds of elements. It is useful to have such class for extensibility reasons. The class manages attributes and methods which are common to every kind of an element.

This class is a parent-class of the class TriangleElement. Its principal tasks are:

- calculate stiffness and mass matrices of elements;
- calculate stresses in elements;
- store and return attributes.

Class TriangleElement

The class implements a triangle element. It inherits methods and attributes from its base-class (Element) and adds to it its own behavior:

- calculate strain matrices $[\mathbf{B}]$ and form function matrices $[\mathbf{N}]$;
- store and return attributes.

Class NodesList

This class manages the list of the structure nodes. Its principal tasks are:

- to manage nodes list;
- to add, delete or return node of the list.

Class Point

The class manages the attributes of the point. This class is a base-class of the class Node. Its principal tasks are:

- to manage point coordinates;
- to store and return its attributes;

Class Node

A node is an attribute of an element. It inherits methods and attributes from the class Point and performs the following task:

- to store and return its attributes.

Class Material

This class is an abstract class for all possible materials. It is used as a base-class for derived classes implementing a particular material. Tasks it performs are:

- to store and return its attributes.

Class ElasticMaterial

This class implements an elastic material. It is derived from the class Material. Tasks it performs are:

- to return material properties: Young's modulus, Poisson ratio, material density;
- calculate matrix $[\mathbf{D}]$ (elastic stiffness tensor).

Class TimeScheme

This class defines time conditions for particular problem. Tasks it performs are:

- sets and returns its attributes: simulation time T and time step Δt .

Class StiffnessMatrix

This class implements the operation of assembly of the structure stiffness matrix $[\mathbf{K}]$.

Class MassMatrix

This class implements the operation of assembly of the structure stiffness matrix $[\mathbf{M}]$.

Class Load

This class is an abstract class for various loads. It is used as a base-class for derived classes implementing particular load. The main task is:

- to store and return attributes, e.g. force value;

Class NodalLoad

The nodal load is a lumped load which acts directly on the structure nodes. It inherits methods and attributes from base-class Load. The main tasks are:

- to compute step nodal loads over all time steps;
- to compute sinusoidal nodal loads over all time steps;

Class DynamicEquation

This class implements all operations associated with solving the dynamic equation. The tasks it performs are:

- to solve the dynamic equation;
- to store and return its attributes.

Class IntegrationScheme

This class is an abstract class for time integration schemes. In this implementation CDS is used, but it can be extended for other kinds of time integration schemes.

Class CDS

This class implements the Central Difference time integration scheme.

Class Math

It implements all necessary mathematical operations with matrices and vectors.

Class ColorBar

It performs calculation of the color palette for contour plots.

Class Graphics

This class implements graphical visualization of the results.

4.2. Main features of classes and objects

4.2.1. Node

The node is the smallest part of the structure. Here we have to describe point coordinates, number of degrees of freedom (DOF), etc. There are five attributes associated with a node: coordinates x (m_X) and y (m_Y), inherited from Point class, number of node (m_NodeNr), array of loads for all nodes (m_load_mas) and numbers of nodes which are affected (m_load_nodes_num) by external loads.

```
//---
class Point
{
private:
    double m_X; // x coordinate
    double m_Y; // y coordinate
public:
    void SetX(double);
    void SetY(double);
    const double &Get_X();
    const double &Get_Y();
};
class Node : public Point
{
private:
    int m_NodeNr;       // node number
    static int *m_load_mas; // array of
                            // effected nodes
    static int m_load_nodes_num; // number of
                           // effected nodes
public:
    void Set_NodeNr(int &nr) { m_NodeNr = nr;
};
    static void Set_load_mas(int *loadmas)
    { m_load_mas = loadmas; };
    static void Set_load_nodes_num(int &num)
    { m_load_nodes_num = num; };
    const int &Get_NodeNr() { return
m_NodeNr; };
    static int *Get_load_mas()
    { return m_load_mas; };
    static int &Get_load_nodes_num()
    { return m_load_nodes_num; };
};
//---
```

All methods in class Node are known as accessors. This means that they do only two operations upon the particular attribute: to set or to get its value. They are used when other classes need to access or to modify the private data of this class.

4.2.2. Element

The class Element manages attributes and methods common to all kinds of elements. Its child class TriangleElement inherits Element and adds its own attributes and methods. The code fragment below describes all attributes and methods in both classes.

```
//---
class Element
{
private:
    int m_elemNr; // element number
    int m_mat_num; // material number
    double m_h; // element thickness
public:
    double **computeElemStiffMatrix
    (TriangleElement *, double **);
    double **computeElemMassMatrix
    (TriangleElement *, double **);
    void computeStress();
    . . . // others set and get accessors
};
class TriangleElement : public Element
{
private:
    int m_firstNode; // first node of element
    int m_secondNode;// second node of element
    int m_thirdNode; // third node of element
    double m_width; // width of element
    double m_S; // square of element
public:
    double **computeBMatrix
    (TriangleElement *, double **);
    double **computeNMatrix
    (TriangleElement *, double **);
    double Side_Length(int, int);
    double Elem_Area(double, double, double);
    double Length_Min(double, double,
double);
    . . . // others set and get accessors
};
//---
```

The class TriangleElement inherits three attributes from the class Element: element number (m_elemNr) the material number of the element (m_mat_num) and thickness of the element (m_h) which is common to all subclasses of class Element.

Attributes describing a triangle element are the three nodes numbers (m_firstNode, m_secondNode, m_thirdNode), triangle width (m_width) and area (m_S).

The methods of the child class TriangleElement compute the strain matrix $[\mathbf{B}]$ and shape functions $[\mathbf{N}]$ which are dependent upon the shape of an element (methods computeBMatrix() and computeNMatrix()). Two methods implementing element stiffness and mass computation are: computeElemStiffMatrix() and computeElemMassMatrix(). Method computeStress() performs calculations of stresses in the element.

4.2.3. Material

Materials used in the structure are described by two classes. The class Material is an abstract class. It is used as a parent-class for all derived subclasses. Its child class ElasticMaterial has three attributes: Young modulus (m_E), Poisson ratio (m_Puas) and material density (m_q). Calculation of the elasticity tensor $[\mathbf{D}]$ is performed in the class ElasticMatrial.

```
//---
class Material
{
public:
    Material();
    ~Material();
};
class ElasticMaterial : public Material
{
private:
    static double *m_E; // Young modulus
    static double *m_q; // material density
    static double *m_Puas; // Poisson ratio
public:
    double **computeDMatrix(int, double **);
    . . . // others set and get accessors
};
//---
```

4.3. Methods

4.3.1 The left-hand side of the dynamic equation

The procedure of numerical integration of the dynamic equation (1) is implemented in the class DynamicEquation by means of computeDisplacements() method. The code is shown below:

```
//---
void DynamicEquation::computeDisplacements()
{
    CDS cd;
    NodalLoad nl;
    MassMatrix mass;
    StiffnessMatrix stiff;
    mass.computeMassMatrix();
    stiff.computeStiffMatrix();
    if (General::Get_Type() == 0)
        nl.computeLoad();
    for (iter=0; iter<time_mom; iter++)
    {
        t += TimeScheme::Get_deltaT();
        if (General::Get_Type() == 1)
            nl.computeLoadSin(t);
        cd.solveCDS(iter);
    }
}
//---
```

The basic task of this method is to assemble and to solve the dynamic equation using central difference time integration scheme. Methods computeMassMatrix() and computeStiffMatrix() compute mass and stiffness matrices, respectively (the code of the method computeStiffMatrix() is shown below). The central difference scheme is implemented by means of solveCDS() method of the class CDS. Methods computeLoad() and computeLoadSin() of the class NodalLoad compute loads vectors.

```
//---
void StiffnessMatrix::computeStiffMatrix()
{
    int i = 0;
    int j = 0;
    int nd = General::Get_NodesOfElement()*
             General::Get_NumberOfDofs();
    int n = General::Get_NumberOfElements();
    double **Ke_Matrix = NULL;
    Element elem;
    ElementList *EL =
ElementList::Get_Start();
    Ke_Matrix = new double*[nd];
    for (i=0; i<nd; i++)
        Ke_Matrix[i] = new double[nd];
    for (i=0; i<n; i++)
    {
        Ke_Matrix =
elem.computeElemStiffMatrix
                (EL->Get_m_elem(),
            Ke_Matrix);
         m_K = construct(m_K, Ke_Matrix,
                         EL->Get_m_elem());
        EL = EL->Get_m_next();
    }
    for (i=0; i<nd; i++)
        delete [] Ke_Matrix[i];
    delete [] Ke_Matrix;
    Ke_Matrix = NULL;
}
//---
```

The method computeElemStiffMatrix(EL->Get_m_elem()) computes the stiffness matrix of the particular element. The method construct() which is called in computeStiffMatrix() performs assembly of the structure stiffness matrix. The argument EL->Get_m_elem() is a pointer to a particular element.

4.3.2. The right-hand side

The computation of the right-hand side (external forces) is performed in two cases, when the load is fixed and when it's sinusoidal. Below is shown computeLoadSin() method implementing sinusoidal nodal load (class NodalLoad).

```
//---
void NodalLoad::computeLoadSin(double t)
{
    computeLoadNodes();
    if (t<1/TimeScheme::Get_f())
    force = m_amplitude*(sin(2*3.14*
            TimeScheme::Get_f()*t));
    else
        force = 0;
    loadVector(force);
}
//---
```

The method computeLoadNodes() sets array of nodes numbers which are affected. The method loadVector(force) sets the load value to a particular

node. The argument "force" is the value of the external load at a given time step.

4.4. Extension possibilities for future development

The structure of the program was designed to satisfy three main features: run-time efficiency, flexibility and extendibility. This program is used as a starting package for future research, so it has to be easy extendible to perform other tasks.

The addition of a new component to the code, e.g. a new finite element shape (rectangle), a new material (plastic) or an external load (elemental) could be adopted easily because of flexible class hierarchy structure:

```
. . .
Element
    TriangleElement
    RectangleElement
. . .
Material
    ElasticMaterial
    PlasticMaterial
. . .
Load
    NodalLaod
    ElementalLoad
```

New classes (e.g., RectangleElement, PlasticMaterial, ElementalLoad, etc.) inherit all the features of their parent-classes and only a minimal number of attributes and methods have to be added.

5. Examples

The geometry and characteristics of a 2D elastic region in which the wave propagation has been analyzed are presented in Figure 3.

The action of a transducer at the boundary of the region is presented by time-varying forces. The geometrical coordinates of the transducer and the time law of the excitation are known. Two cases of excitation forces (step force and harmonic force) have been analyzed.

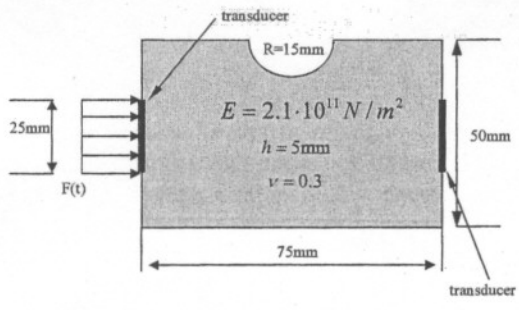

Figure 3. Elastic media

6. Results

The structure meshed by triangle elements is presented in Figure 4. Results presented in figures below are shown in 3D case, but actually the problem was solved in plane. Figure 5 illustrates "frozen" view of the propagating wave in terms of equivalent stresses in the structure affected by step load. Equivalent stresses obtained by applying the harmonic load are given in Figure 6.

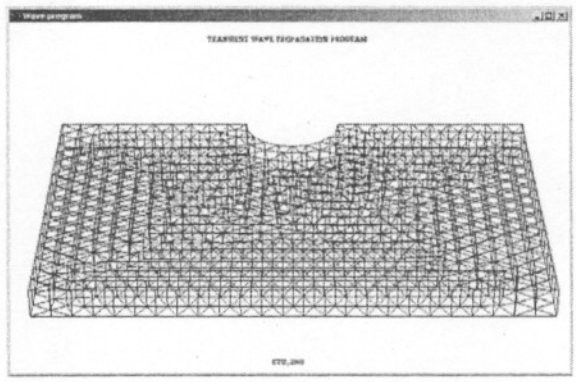

Figure 4. Media meshed by triangle elements

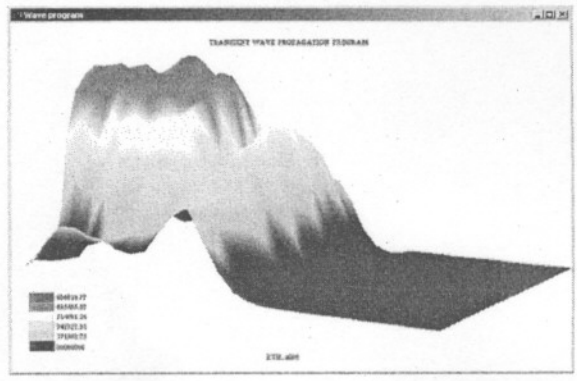

Figure 5. Equivalent stresses of the structure after 120 time steps

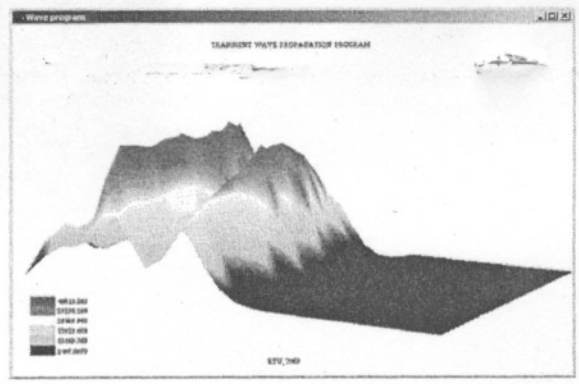

Figure 6. Equivalent stresses of the structure (sinusoidal shape of excitation forces) after 120 time steps

7. Conclusions

An object-oriented approach for creating finite element program for the wave propagation analysis in an elastic environment has been described. Several programming approaches, as well as, the main OO advantages have been discussed.

The OO programming adopted for the FE method allows the easy development of program extensions. The example, presented in the transient wave propagation program shows the advantages of the code flexibility and efficiency.

Visualization facilities in OpenGL, especially in 3D, give many possibilities for presenting results in good view with fast execution time. All reasons tend that OO programming approach is one of the best in FE applications development.

References

[1] **R. Barauskas**. On space and time step sizes in ultrasonic pulse propagation modeling. *European Conference on Computational Mechanics, Cracow, June* 2001, 26-29.

[2] **S. Commend and T. Zimmermann.** Object-Oriented Nonlinear Finite Element Programming: a Primer. *Advances in Engeneering Software, Vol.*8, 2001, 611-628.

[3] **J.T. Cross, I. Masters and R.W. Lewis**. Why you should consider object-oriented programming techniques for finite element methods. Object-oriented programming techniques. *International Journal of Numerical Methods for Heat & Fluid Flow, Vol.*9, *No.*3, *MCB University Press*, 1999, 333-347.

[4] **Y. Dubois-Pelerin, T. Zimmermann**. Object-oriented finite element programming 3. An efficient implementation in C++. *Computer Methods in Applied Mechanics and Engineering, Vol.*108, 165-183.

[5] **Z.Q. Feng**. 2D or 3D frictional contact algorithms and applications in a large deformation context. *Communications in Numerical Methods in Engineering, Vol.*11, 409-416.

[6] **J. Liberty**. C++ in 21 Days. *Second Edition. Sams Publishing, Indianapolis*, 1997.

[7] **R.I. Mackie**. Object Oriented Methods and Finite Element Analysis. *Saxe-Coburg Publications*, 2002.

[8] **P. Mentrey, T. Zimmermann**. Object-oriented nonlinear finite element analysis – application to J2 plasticity. *Computers and Structures, Vol.*49, 767-777.

[9] **H. Ohtsubo, Y. Kawamura, A. Kubota**. Development of the object-oriented finite element modelling system – modify. *Engineering with Computers, Vol.*9, 187-197.

[10] **R.V. Pidaparti, A.V. Hudli**. Dynamic analysis of structures using object-oriented techniques. *Computers and Structures, Vol.*49, 149-156.

[11] **E. Stensrud, I. Myrtveit**. Measuring productivity of object-oriented vs procedural programming languages: Towards an experimental design. *Oslo, Norway*.