An Object Oriented FEM System

Alexandru Telea

12th December 1996

Abstract

This paper gives a short presentation of an object-oriented FEM system and emphasizes on the advantages of object orientness in the task of building such a system. The model presented here has been implemented as a C++ 'class library' and has resulted in a fully operational FEM system.

1 Goals Accomplished So Far

The C++ FEM system currently available is able to perform computations on arbitrary two-dimensional domains. It supports convection-diffusion, Navier and Navier-Stokes PDEs, conjugated gradient and BiCGSTEP solvers and SSOR and ILU(0) preconditioners. All problems can be time dependent (instationary) or stationary. The supported boundary condition types include Dirichlet and Neumann boundary conditions. Three-point and six-point triangular elements are supported together with two different, transparently interchangeable mesh generators. All phases of a FEM simulation are supported: problem/domain definition, boundary conditions definition, problem solving and simulation visualization.

The system works completely in terms of *objects*: points, curves, surfaces, domains, problems, cameras, scalar/vector fields. This model allows a user to specify, solve and visualize a problem with an effort comparable to writing a 'batch' file for a simulation package, but with the full advantage of a system open to programming, upgrading and user interactivity.

2 System Structure

The general structure of the system is that of a *class library*. Several base classes are provided for establishing an interface between the FEM universe and the user. These base classes can be 'customized' in various ways in order to provide specific functionality.

As a general rule, the system is *open* to the addition of new *features* to classes or to the modification/specialization of existing features. This is a very important and not trivial requirement, especially if run-time flexibility and interactivity is a goal.

The next sections will present the class set and outline the reasons for which the present structure was adopted. The class set is divided into three parts, according to a common functionality of the classes of a part. First part describes *geometrical classes*, i.e. classes that describe the geometry of the problem's universe. The second part describes *problem definition classes*, i.e. the classes mainly used to state a problem. The third part contains *interaction classes*, mainly concerned with visualization and user interaction.

3 Geometrical Classes

A FEM problem is defined over a physical domain. The description of this domain is the task of geometrical classes. These classes are very similar to standard 2D geometrical classes from computer graphics systems. There are however important internal differences that have to do with efficiency and data representation required by the other FEM classes.

3.1 Class USERPOINT

A USERPOINT is the simplest geometrical token available for a problem domain definition. It is used as a control point for the curves that will eventually build the domain's boundary. A USERPOINT is basically a two dimensional point having an extra 'coarseness' value which determines the mesh refinement around the domain's area where the point is. A problem definition will therefore proceed with the creation of some USERPOINTs that will determine the shape of the computational domain. A USERPOINT is therefore a POINT plus a coarseness value (see

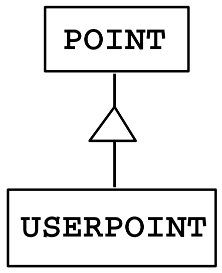

Figure 1: USERPOINT class

figure 1). POINT is a base class for USERPOINT which contains the point's *x* and *y* coordinates.

3.2 Class USERCURVE

A USERCURVE is a two-dimensional curve connecting two USERPOINTs. It is the next construction element which can be used to build a computational domain, after the USERPOINT. A USERCURVE is determined by its type (i.e. the kind of curve it is, like circle arc, ellipsis arc, straight line, polyline, etc) and a set of control USERPOINTs which determine its shape and location. For example, a USERLINE is determined by two USERPOINTs, a USERARC is determined by three USERPOINTs (which can be three points on the arc or two points at the arc's ends and a third being the center of the circle the arc belongs to), a POLYLINE is determined by the sequence of all points it contains, and so on.

All curve classes are derived out of an abstract class USERCURVE which contains all common curve features (like inquiring about control points, moving a control point, etc). USERCURVE has no constructor, since we can't construct an abstract shapeless curve. The other classes are derived out of USERCURVE and add only the constructors.

A USERCURVE internally consists of a uni-dimensional meshing of the curve. Points are created on the curve in between the control USERPOINTs. We call this process 'meshing' a curve. The user should generally not access the internal representation but manipulate the whole problem in terms of high-level objects. However sometimes there is a need for a point-level examination of USERCURVEs, therefore the internal meshing data is available for reading.

Classes derived out of USERCURVE should (ideally) not add any other methods

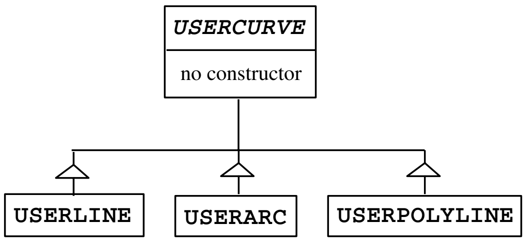

Figure 2: USERCURVE base class and derived classes

but the constructor, since a 'generic' curve should be manageable via the USERCURVE interface.

USERCURVE is used also to implement boundary conditions. The user can easily prescribe a condition type and value over a USERCURVE. Condition values are actually C++ callback functions that are supplied by the user.

3.3 Class SURFACE

A SURFACE is essentially a two-dimensional arbitrary region of the plane defined as the zone enclosed by a set of USERCURVEs. The list of USERCURVEs defining a SURFACE is called the surface's *contour*. A USERCURVE can be added to a surface with a plus or minus sign, this sign determining the way the curve is 'run' when the contour is scanned. This allows using 'fake' curves to describe a surface with holes, by using a curve twice (with different signs) to connect the hole's contour to the outer contour. Of course, a SURFACE's contour should be well-defined (i.e. should not have self intersections and should close properly).
A SURFACE is meshed in order to produce a list of ELEMENTs. An ELEMENT

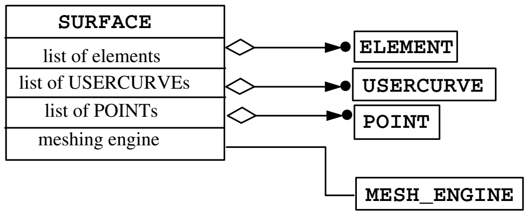

Figure 3: SURFACE class and its main members

is the class representing the mathematical finite element. It basically consists of a (fixed size) set of POINTs. When meshing the SURFACE, new points are created. The SURFACE will store also a list of all POINTs being created at its meshing. A SURFACE has a reference element size which determines the size of the elements produced when meshing it. This is a global mesh refinement control, seen as opposed to the local refinement given by the USERPOINT coarseness.

There are many ways to mesh a SURFACE and produce a list of ELEMENTs and a list of POINTs. We have encapsulated the meshing phase as a class called MESH_ENGINE. A MESH_ENGINE is therefore a class that takes a SURFACE and meshes it. Since (run-time) flexibility is a major issue, we have decided to have the SURFACE and the MESH_ENGINE as two *separate* notions. The connection is that a SURFACE *uses a* MESH_ENGINE, which practically means that a SURFACE object keeps a reference to a MESH_ENGINE object which it will use when meshing itself. The MESH_ENGINE has a thin interface, consisting (so far) only in a constructor and the *mesh()* method. This design issue is of a fundamental importance and will be discussed later in its more general aspect. The separation of the SURFACE from MESH_ENGINE allows us to attach/detach or replace a MESH_ENGINE from a SURFACE at run-time. We can therefore use different MESH_ENGINEs having different space/time performances transparently. The current implementation features two MESH_ENGINEs that use totally different surface meshing policies and internal data representations and both produce

three-point triangular elements:
Besides the geometrical information, a SURFACE carries also FEM data like phys-

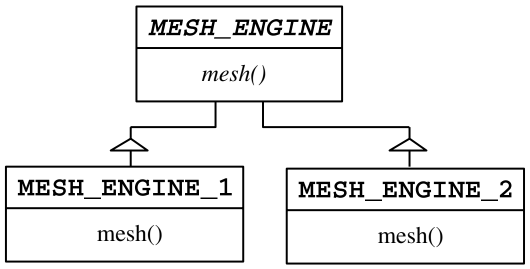

Figure 4: MESH_ENGINE class hierarchy

ical surface properties (diffusivity, source and convection terms, etc) given as user-written callback functions defined over the surface.

4 Problem Definition Classes

Problem definition classes encapsulate more specific FEM data and behaviour. They are one layer above the geometrical classes, i.e. the *use* geometrical classes for their definition. Actually almost all the FEM code is encapsulated in these classes.

4.1 Class DOMAIN

A DOMAIN object describes, as its name says, the computational domain of a FEM problem. Basically a DOMAIN is a collection of adjacent two-dimensional surfaces that share common USERCURVEs. Describing a FEM problem's DOMAIN as a collection of SURFACEs has several advantages:

- different surface properties (e.g. diffusivity, source and convection terms, viscosity, etc) can be prescribed on different parts of the computational domain. This leads to a natural domain decomposition in surfaces.
- complex geometries can require different mesh refinements over different areas. This can be easily done by choosing different reference element sizes for different surfaces in a domain.

When a DOMAIN is built, its component SURFACEs are assembled, global ELEMENT and POINT lists (containing all ELEMENTs and POINTs over the DOMAIN) are created and some topological data (like neighbour points of each point and neighbour elements of each element) are created. This data will be heavily used by the FEM code, so attention has to be paid to its storage and access speed.

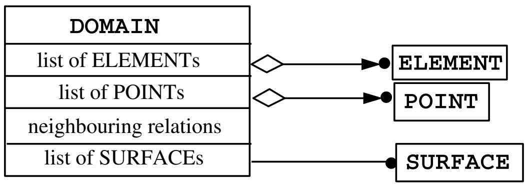

Figure 5: DOMAIN class

Moreover, in the phase of assembling SURFACEs into a DOMAIN, some new POINTs are created. Midpoints of the 6-point elements (or Taylor-Hood elements) are *not* created during SURFACE meshing, in order to keep the meshing algorithm simple and fast. If the DOMAIN contains such elements, these midpoints are created now and inserted into all the elements, which practically become 6-point triangles out of 3-point triangles. Node renumbering is also done in this stage in order to obtain a smaller bandwidth of the FEM system's matrix. This is done by a statically built-in code in the DOMAIN class, but it could be done by a 'renumbering engine', allowing different renumbering schemes to be interchangeable transparently and at run-time.

4.2 Class PROBLEM

PROBLEM is the highermost class in the FEM library. It describes a finite element problem, posed on a given DOMAIN, having some given initial conditions. It also stores information concerning whether the problem is time dependent or not. Some other FEM specific data is stored in the PROBLEM class (e.g. stiffness matrix, solution vector, stiffness matrix builder, solver, etc).

The specific FEM data used by PROBLEM are also implemented in an object-

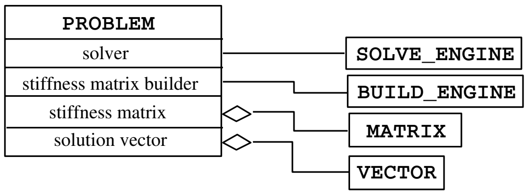

Figure 6: PROBLEM class

oriented fashion. The stiffness matrix is encapsulated in a general MATRIX class and the solution vector uses a VECTOR class.

A PROBLEM is fundamentally characterized by its *type* (e.g. diffusion, Stokes, etc). Almost all the problem-specific part is encapsulated in the concept of a *builder engine*. A BUILD_ENGINE (in OO terms) is a class that is responsible with build-

ing the PROBLEM's stiffness matrix, which is actually the action giving the specific of a FEM problem. We used here the same concept as for mesh engines, i.e. separating the matrix building concept from the problem one. This allows us to have a PROBLEM object and customize it with different builder engines, practically making it a different problem each time.
The same is true for the solver. The PROBLEM object uses a SOLVE_ENGINE

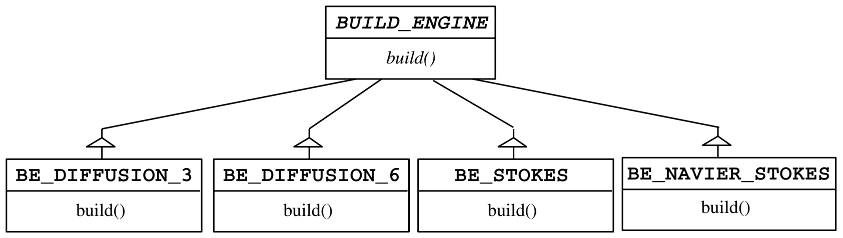

Figure 7: BUILD_ENGINE class hierarchy

which is basically a solver taking the stiffness matrix and filling the solution vector. Several solvers are available to be connected to a PROBLEM so the user can choose the solver that best corresponds to its PROBLEM object. There are currently two solvers implemented, one for symmetric matrices and the other one for asymmetric matrices. The SOLVE_ENGINE contains internally a preconditioner engine, which preconditions the stiffness matrix prior to solving. Its interface is again based on the interchangeable engine concept.

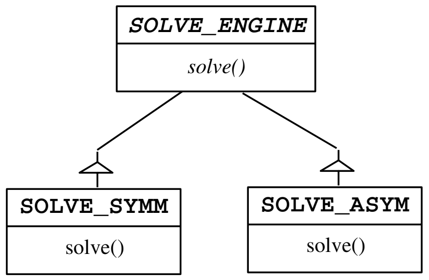

Figure 8: SOLVE_ENGINE class hierarchy

5 Interaction and Visualization Classes

The last part of the FEM system is concerned with user interaction and visualization. The main concept is the CAMERA class. A CAMERA is basically a window-based view of a set of FEM objects (e.g. USERCURVEs, DOMAINs, PROBLEMs, etc). Besides displaying the objects, a CAMERA will allow the user to interact with them (e.g. select an object, move it, point to a location inside it, etc).

All the above functionality is implemented as a class hierarchy having the CAMERA as base class providing mainly the interface for adding/removing objects to/from a camera and for changing the viewpoint by translations, rotations and scalings. The next level is GL_CAMERA, a class which defines a supplementary interface part, based on the OpenGL graphics library. While CAMERA is a system-independent class, GL_CAMERA is system-dependent class. GL_CAMERA is also an abstract class.

There are two derivation branches of GL_CAMERA. The first is AUXCAMERA,

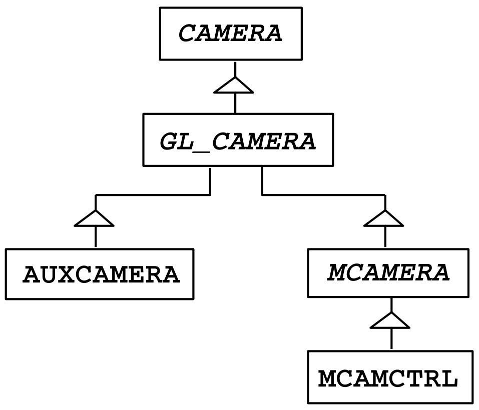

Figure 9: CAMERA class hierarchy

a simple fully functional camera based on the AUX extension to OpenGL. It features a window driven by the window manager that is able to display FEM objects and allow selection and location. Its disadvantage is that only one AUXCAMERA can be opened per application, due to the structure of the internal loop of the AUX system.

The other derivation is MCAMERA, a camera based on OpenGL and the Motif user interface. the purpose of the MCAMERA is to be added as a widget in a more complex Motif application. Practically, it is a 'FEM visualization and interaction Motif widget'. A more sophisticated and easier to use camera is MCAMCTRL, which is a standalone ready-to-use camera having Motif controls for viewpoint positioning. Applications can take full advantage of the Motif interface by using multiple MCAMERA objects that will open several windows each and allow independent visualization of different or the same FEM objects.

6 A Design Issue: Object Composition versus Class Composition

An essential design issue has been outlined a couple of times during this paper. Basically it can be seen as having two ways of adding features to a (C++) object. The first way is class composition, that adds features (e.g. methods or data members) by *inheritance*. The second is object composition, adding features by storing a *reference* to the new feature in the object.

Object composition (sometimes called delegation) has some major advantages for a high-level object-oriented design:

- first of all, it allows adding features *per object instance* rather than per class. In a system having multiple independently varying features this keeps the number of classes at a minimum.
- it allows changing a feature at run-time rather than at compile time, thus making the system much more flexible. The result will be that object 'types' will seem to partially change at run-time.
- it decouples the feature design from the object's design even more than inheritance does.
- it removes the need for expensive and problematic virtual inheritance class hierarchies

The featured example in this paper is the engine concept. The main disadvantage of object composition is run-time efficiency. However the use of virtual functions in a class composition is not (generally) significantly faster than object composition, which can be seen in the end as a run-time modifiable virtual function mechanism.