FEView: An interactive visualization tool for finite elements

Yao Zheng.a,*, Roland W. Lewis.a, David T. Gethin.b

a Department of Civil Engineering, University College of Swansea, University of Wales, Swansea, SA2 8PP, UK
b Department of Mechanical Engineering, University College of Swansea, University of Wales, Swansea, SA2 8PP, UK

Abstract

This paper presents a finite element visualization facility, FEView, which has been implemented based upon an object-oriented graphics library. The visualization tool works as an external module to an interactive program Geomview for viewing and manipulating geometric objects. The graphical user interface has been built on top of the Forms Library, a graphical user interface toolkit for Silicon Graphics workstations.

A finite element mesh can be considered as a collection of faces with edges, wire frame, or point cloud, and the corresponding numerical results gained through finite element analyses can be visualized via color shading and field icons (such as arrows) on the geometric shapes. Also, a scalar field can be represented as a weather map to highlight color shading domains with scalar values falling into the range of interest. Numerical results for two-dimensional cases can be shown with three-dimensional effects by using values of the scalar field. FEView provides animation control over single frame stepping and adjustable speed playing. It has been equipped with geometry operation functionality, in which a particular part of an object can be obtained by specifying material indices, element numbers, and cutting boxes. In local analysis mode, FEView is able to provide local information about finite element objects by picking up the position of interest via mouse manipulation.

1. Introduction

The finite element method has become a very important analysis tool in many industries as a result of the increase in computer power and the progress in numerical computation technologies. The analysis produces large data sets, and the effective visualization of these data sets has become an important task to assist with interpretation and analysis in turn. For real industrial applications, the geometries involved are normally complex and the visualizers for 3D cases are required to be able to represent the geometries and the corresponding numerical results for physical interpretation. For instance, in the numerical modelling of casting problems, the visualizers are

*Corresponding author.
desired to be capable of presenting temperature (scalar), flow (vector) and stress (tensor) fields in 3D configurations. In the development of mesh generation, a visualization tool is necessary to enable the developers to evaluate resulting meshes in various aspects, for example, to ensure their detailed correctness and accuracy. Finite element applications demand sophisticated user-friendly visualization tools [1–4]. The present paper describes a finite element visualization tool which has been developed based on an object-oriented graphics approach. Finite element meshes can be considered as collections of faces with edges, wire frames, or point clouds, and the corresponding numerical results gained through finite element analysis can be visualized via color shading and field icons (such as arrows) on the geometric shapes.

2. Display engine

2.1. Pipelines

FEView works as an external module to Geomview, an interactive program for viewing and manipulating geometric objects [5, 6]. It uses Geomview as a display engine, therefore the functionalities which come with Geomview, such as object rotating, translating, zooming, and scaling, are available. FEView communicates with Geomview through GCL (Geomview Command Language), and informs Geomview of the new geometric object shape at each time, so the object appears to evolve with time in the Camera window of Geomview. Also, the external module can respond to mouse and keyboard events that take place in a Geomview window.

When Geomview invokes FEView, it creates pipes connected to the module's standard input and output. These pipes are like files except that they are used for communication between programs rather than for storing data on a disk. Geomview interprets anything that the external module writes to the standard output of the module. Likewise, if the external module requests any data from Geomview, this information is to be provided to its standard input (see Fig. 1). FEView is the same as other external modules and appears in the Modules browser in the Main panel of Geomview (Fig. 2).

![Diagram of pipelines between FEView and Geomview](image)

Fig. 1. Pipelines between FEView and Geomview.
Fig. 2. The main panel of Geomview.

2.2. Geomview

Many aspects of geometry display and interaction in a program are independent of the actual geometric content and can be collected together in a single piece of software that can be used in a wide variety of situations, Geomview. External modules are then implemented concentrating on the desired algorithms and leaving the display aspects to the presentation software. Geomview can be used as a standalone viewer for static objects or as a display engine for other programs which produce dynamically changing geometry. Geomview supports the following geometry objects: polygons with shared vertices, quadrilaterals, rectangularly connected meshes, points and lines, and Bezier surface patches of arbitrary degree including rational patches. Object hierarchies can be constructed with lists of objects and instances of object(s) transformed by one or many $4 \times 4$ matrices. Arbitrary portions of changing hierarchies may be transmitted by creating named references. Additionally, Geomview can display objects in hyperbolic, spherical and Euclidean space.

The main purpose of Geomview is to display objects whose geometry is given, allowing interactive control over details such as point of view, speed of movement, appearance of surfaces and lines, and so on. Geomview can handle any number of objects and allows both separate and collective control over them. The software also provides interactive control over motions, appearances (including lighting and shading), picking on an object, edge or vertex level, and adding or deleting objects, through direct mouse manipulation, control panels and keyboard shortcuts. By default, when the user starts up Geomview, three windows appear: the Main panel, the Tools panels, and one Camera window. Fig. 3 shows an initial layout after some geometries have been loaded.

Geomview can display an arbitrary number of objects simultaneously, and the TARGET browser in the Main panel displays a list of all the objects that the presentation software currently
knows about. This browser has a line for each object that has been loaded. A further object is called “World” and corresponds to all the currently loaded objects, treated as if they were one object. To manipulate an object, it must be selected from the TARGET browser. Motions are applied by appropriate mouse control input in the Camera window. There are several different motion “modes”. The MOTION MODE browser in the Main panel indicates the current mode, which can be changed by selecting a new one from the MOTION MODE browser. Also, the motion mode can be controlled through the Tools panel, which is to be outlined later in Section 4.2.

3. Object-oriented graphics library

3.1. Object-oriented technology

Object-oriented technology has a number of features which underpin the development of high quality software. It contains methods, processes, and tools used to design software from objects. Objects are capsules of behavior (functions) and state (data), whose internals are hidden from other objects. The object-oriented concept has begun to appear in many new applications (e.g. [7–9]). The following is a brief outline of basic object-oriented principles and mechanisms, which represent
the necessary characteristics of an object-oriented software system and these follow as a consequence of reviewing literature on the topic.

- **Abstraction** – This is the human cognitive ability to concentrate on the relevant aspects of an object while ignoring non-relevant aspects. It allows the designer to deal with high-level concepts first and then to consider the details if necessary. Object-oriented languages support abstraction through classes and messages.

- **Modularity** – Modularity is the process of subdividing a complex systems into components and modules. In object-oriented approaches, modules are defined to be **Object** and **Class**.

- **Encapsulation (Information Hiding)** – This is the process of hiding behavior details from the end user, and supports effective maintenance and evolution of the module. Thus, internal design of a module can be enhanced without disturbing its interfaces.

- **Hierarchy** – This is the method used to divide the complex systems and the ability to connect the different levels of system abstractions within modules.

- **Inheritance** – It is a relation between classes that allows for definition and implementation of one class to be based on that of other existing classes.

- **Typing** – This is the enforcement of the class of an objects, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

- **Dynamic Binding** – Dynamic binding waits until execution time before binding variables to types. The advantage is that a variable may represent one of many different objects from different classes.

- **Polymorphism** – A variable that can represent more than one class of object is polymorphic.

- **Persistence** – Persistence is the property of an object through which its existence transcends time and/or space.

Abstraction, modularity, encapsulation, hierarchy and inheritance can be used to identify objects and classes of a software, and also help to classify objects and classes that belong to the base system (kernel) or non-base system. Other mechanisms such as polymorphism, typing and dynamic binding allow the system to be more flexible, extendible and adaptable.

### 3.2. Object-oriented graphics library

Geomview has been constructed based upon on Object-Oriented Graphics Library (OOGl) [5, 6]. The OOGl routines are a high-level set of modules which allow the user to specify and display geometry. The generic OOGl class is called a **Geom**, which is defined in Fig. 4. Its subclasses include geometric primitives such as **PolyList** (polygons), **Vect** (points and lines), **Bezier** (Bezier surface patches), and **Mesh** (rectangularly-connected meshes); and organizational objects such as **List**, **TList** and **Inst** (for instancing geometry). Each type of geometric object, whether low-level or high-level, comes equipped with methods, i.e., a standard set of operations. The methods include: **Bound**, **Create**, **Copy**, **Delete**, **Save**, **Load**, **Pick**, and **Draw**. A virtual function table is also described in Fig. 4. This style of programming, which combines data structures with operations acting on these data structures, is known as object-oriented programming (e.g. [10, 11]).
Generic OOGL class

```c
struct Geom { /* common data structures for all Geom's */
    REFERENCEFIELDS /* magic, ref_count, handle */
    struct GeomClass *Class;
    struct Appearance *ap;
    Handle *aphandle;
    int geomflags;
};
```

Virtual function table

```c
struct GeomClass {
    /* General Methods */
    GeomClass *super; /* superclass of this class */
    GeomNameFunc *name;
    GeomMethodsFunc *methods;
    GeomMessageFunc *message;
    GeomGetFunc *get;
    GeomCreateFunc *create;
    GeomDeleteFunc *delete;
    GeomCopyFunc *copy;
    GeomReplaceFunc *replace;
    GeomExtFunc **extensions; /* Extension methods live here */
    GeomFLoadFunc *fload;
    int n_extensions; /* Size of extensions[] array */
    GeomFSaveFunc *fsave;

    /* Geometric Methods */
    GeomPositionFunc *position;
    GeomTransformFunc *transform;
    GeomTransformToFunc *transformto;
    GeomEvertFunc *evert;
    GeomBoundFunc *bound;
    GeomEvalFunc *eval;
    GeomDiceFunc *dice;
    GeomSubdivideFunc *subdivide;

    /* Picking methods */
    GeomPickFunc *pick;
    GeomBoundSphereFunc *boundsphere;
    GeomIterateFunc *iterate;
    GeomAppendFunc *append; /* Append new item to hierarchy object */
    GeomScanFunc *scan;

    /* Graphics Methods */
    GeomFacingFunc *facing;
    GeomDrawFunc *draw;

    /* Communications methods */
    GeomExportFunc *export;
    GeomImportFunc *import;
    GeomUnexportFunc *unexport;
};
```

Fig. 4. Data structures of OOGL object.
FEView has been developed to normally send geometry shapes in OOGL file formats to Geomview. However, in a local analysis state, it communicates with Geomview in both reading and writing directions. In this case, FEView calls the OOGL Lisp library to parse and act on the messages that FEView reads from Geomview.

4. FEView

4.1. Outline of FEView

The master panel which itemized the features incorporated into FEView is shown in Fig. 5. Starting up FEView will bring up an moving logo “Welcome to FEView” in the Camera window (Fig. 6), and initializes some variables with default settings. An outline of FEView is itemized as following.

- FEView can deal with two-dimensional and three-dimensional applications. In Quasi 3D mode, a two-dimensional scalar field can also be shown with three-dimensional effects using the scalar values as the third geometry coordinates.
- Static and dynamic results can be treated, along with efficient file picking ability, through the Static Control panel and the Animation Control panel, respectively. Animation can be played step-by-step forward and backward, or at an adjustable speed.

![Fig. 5. Master panel for FEView.](image-url)
Fig. 6. Initialization of FEView.
A particular part of an object with finite elements can be obtained by specifying material indices, element numbers, and cutting boxes via the Domain Control panel.

Through the Feature Control panel, the features of the representation can be controlled. Apart from scalar fields, the vector and tensor fields can also be visualized by using field icons.

Using the Column Control panel, it is easy to shift between different fields when finite element results are being visualized.

For contours, two shading schemes, i.e., continuous and graded coloring, have been introduced and can be controlled via the Shade Control panel. Both of them have color and monochromatic versions.

Local information about finite element objects can be output in the Local Analysis panel by pointing to the position on the objects.

4.2. Motion control

As Geomview serves as a display engine for FEView, all functionalities of motion control of Geomview are retained during the execution of FEView. There are six different motion modes to represent object and viewpoint motion: Rotate, Translate, Cam Fly, Cam Zoom, Geom Scale, and Cam Orbit. The motion mode can be controlled through the Tools panel by means of the mouse (Fig. 7).

A mouse motion control can be applied by holding down either the left or middle mouse button with the cursor in a camera window and moving the mouse. Most of the modes have inertia, which means that if the user lets go of the button while moving the mouse, the motion is to continue. Most of the mouse motions have a slow motion version which the user can get by holding down the shift key while doing the motion as usual. And this is useful for finer control.

It should be mentioned that Geomview uses the glass sphere model for mouse-based motion. This means the user is supposed to think of the object as being inside an invisible sphere and the mouse cursor is a gripper outside the sphere. When the user holds down the left mouse button, the gripper grabs the sphere; when the user lets go of the button, the gripper releases the sphere. Moving the mouse while holding the button down causes the sphere (and hence the object) to move in the same direction as the mouse.

The buttons Stop, Look At, Center and Reset on the Tools panel perform actions related to motions but do not change the current motion mode (Fig. 7). The aforementioned six mouse motion modes can be outlined as follows.

- Rotate – The left mouse button rotates the objects in such a way that the axis of rotation is parallel to the camera view plane, and is perpendicular to the direction of motion of the mouse. Meanwhile the middle mouse button rotates the object about an axis perpendicular to the view plane.
- Translate – The left mouse button translates the objects in the direction of mouse motion whereas the middle one translates the objects along an axis perpendicular to the view plane.
- Cam fly – It is a crude flight simulator that lets the user fly around the scene.
- Cam orbit – This mode lets the user rotate the current camera around the current center.
- Cam zoom – It lets the user change the current camera’s field of view with the mouse.
• Geom scale – This mode lets the user enlarge or shrink a geometry object. Moving the mouse while holding down the left mouse button scales the object either up or down, depending on the direction of mouse motion.

4.3. Appearance

Geomview uses a hierarchy of appearances to control the way objects look. This includes many aspects such as color, lighting, material properties, and so on. Appearances work in a hierarchal manner: If a certain appearance property, e.g. face color, is not specified in a particular object’s appearance, that object is drawn using that property from the parent appearance. Every geometry object in Geomview has an appearance associated with it. There is also an appearance associated
with the “World” geometry, which serves as the parent for the appearance of each individual object. Finally, there is an global “base” appearance, which is the parent of the “World” appearance.

FEView specifies some aspects of the object appearance automatically, while others such as basic color settings and color schemes can be controlled through the panels of FEView. The objects are visualized in the following four shading modes:

- **Constant** – Every face of the object is drawn with a constant color which does not depend on the location of the face, the camera, or the light sources.
- **Flat** – Each face of the object is drawn with a color that depends on the relative location of the face, the camera, and the light sources. The color is constant across the face but may change as the face, camera, or lights move.
- **Smooth** – Each face of the object is drawn with smoothly interpolated colors based on the normal vectors at each vertex. If the object does not contain per-vertex normals, this has the same effect as flat shading. If the object has reasonable per-vertex normals, the effect is to smooth over the edges between the faces.
- **CSmooth** – Each face of the object is drawn with exactly the specified color(s), independent of lighting, orientation, and material properties. If the object is defined with per-vertex colors, the colors are interpolated smoothly across the face; otherwise the effect is the same as in Constant shading style.

5. Finite elements

5.1. Finite element meshes

This visualization tool can deal with two-dimensional and three-dimensional applications. At present, all the discretized values of scalar fields are represented using linear interpolation, no matter whether linear or nonlinear elements are involved. The numerical results are visualized in several styles such as a collection of surface patches with color shading except vector and tensor fields, in which case field icons are to be used.

Therefore, in principle, FEView is capable of handling numerical results for all types of elements. At present, for two-dimension problems, five types of meshes can be represented by this tool including:

1. 3-noded triangles;
2. 4-noded triangles;
3. 6-noded quadrilaterals;
4. 8-noded quadrilaterals; and
5. 9-noded quadrilaterals.

In three-dimensional cases, the following seven types of solid meshes can be visualized by FEView:

1. 4-noded tetrahedrons;
2. 10-noded tetrahedrons;
3. 6-noded pentahedrons;
4. 15-noded pentahedrons;
5. 8-noded hexahedrons;
(6) 20-noded hexahedrons; and
(7) 21-noded hexahedrons.

5.2. Geometry operations on FE meshes

It is often required for the computational domain to be split to enable viewing of an interesting part of the domain. Geometry operations on FE meshes have been implemented, so that the target
geometry object (e.g. a collection of surface patches) can be identified by specifying material indices, element numbers, and a cutting box (Fig. 8), which can be selected via the Domain Control panel (see Fig. 9). Fig. 10 illustrates some examples of domain control: (a) is the whole computational domain; (b) is a mesh with specified material indexes while (c) is a mesh further specified by a cutting box; And (d) shows a specified element and its neighbour. Also, the elements can be shrunk, which give a new style of representation (Fig. 11). In the three-dimensional case, shrinking enables the user to inspect parts internal to the solid objects to a certain extent.

The above geometry operations take time to execute, where several frames sometimes have same geometry configuration. In order to avoid repetition, geometry configuration checking is carried out for every draw command. To enable efficient programming, a data structure named GeomConfig (Fig. 12) has been introduced, which records the geometry operation information.

Fig. 10. Examples of domain control.

Fig. 11. Elements shrinking.
typedef struct {
    char fe_meshfile[512];
    int fe_mstep, fe_idmesh;
    int numb_dim, numb_points;
    int numb_elements, i_element_show;
    int numb_materials, i_material_show, i_material_hide;
    int numb_columns, i_column_colors, i_column_height;

    int numb_faces_work;

    int material_option, element_option, box_option, box_sub_option;
    float dc_xb1, dc_xb2, dc_yb1, dc_yb2, dc_zb1, dc_zb2; /* cutting box */

    int dimflag, stateflag;
    int domainflag, shrinkflag, nocomfaceflag;
    int appearanceflag, meshformatflag, focusflag, saveflag;
} GeomConfig;

Fig. 12. Data structure of geometry configuration.

It should be mentioned that the object relating to a mesh can be given directly by a face-based description in a certain file format. In this case, the geometry is not to be done, since this description is already obtained in a similar way.

6. FE mesh representation

6.1. Basic description

The finite element objects need to be represented in different color schemes. A mesh without related numerical results is visualized in some basic colors, which can be specified by using a colorwheel in the Basic Color panel. The mesh can be viewed by faces with edges, which is a common way of representation (Fig. 13). The mesh can also represented in other ways such as wireframe and point cloud. The wireframe and point cloud are helpful in the development of mesh generators, since the quality of generated meshes in some aspects can be evaluated by inspection of these views [12].

6.2. Wire frame

Fig. 14 illustrates a wireframe of a three-dimensional mesh. Although the representation may not give a clear impression, it provides an indication of the mesh density at different locations. When the mesh is associated with a set of numerical results, it can be visualized in continuously changing colors (e.g. Fig. 15).

6.3. Point clouds

A point cloud consists of a group of dots at the nodal points (see Fig. 16). The three-dimensional point clouds give the user depth-wise information about the nodal information. Again, if the mesh
is assigned with a set of numerical results, it can be visualized in continuously changing colors, which represent nodal values (e.g. Fig. 17).

7. Scalar field visualization

7.1. Color schemes

For scalar fields, a general way to visualize the results is by in color shaded contours. In FEView there are two version of continuous color scheme: grey scale and color scale. The former can be used in the monochromatic environments.
Fig. 15. A wireframe of a 2D FE mesh with numerical results.
Let $V$ be a normalized value in a scalar field. In the HSV (Hue, Saturation and Value) color system [13], $V$ can be assigned a color with

$$h = 0.0, \quad s = 0.0, \quad v = 0.5 \, V,$$

in the grey scale. While in the color scale it can be assigned a color with

$$h = \frac{2}{3} (1.0 - V), \quad s = 1.0, \quad v = 1.0$$

Since there is a standard procedure to map $(h, s, v)$ values into $(r, g, b)$, it is easy to define these colors in the RGB (Red, Green and Blue) mode, with which the geometry objects are described for Gomview.
Fig. 17. Point clouds of a 2D FE mesh with numerical results.
Fig. 18. Quasi 3D representation of a shading contour.
Fig. 19. 3D shading contour in continuous color mode for dynamic FE results.
Fig. 20. 3D shading contour in grading color mode for dynamic FE results.
Fig. 21. Weathermap of dynamic FE results.
The above color schemes define continuous colors. It should be mentioned that there is a way to highlight the values within a certain range, in which three particular colors can be used for this purpose. It is called grade color scheme. Also, there are two version, one for the grey scale, the other for the color scale.

7.2. Geometry dimension

The 2D and 3D meshes, whether they are associated with numerical data, can be visualized in 2D and 3D views, respectively. It is worth mentioning that a 2D mesh with a scalar field can be represented in a 3D view as a Quasi-3D format. In this case, the values in a scalar field can be utilized as the third geometry coordinates. Fig. 18 shows a quasi-3D representation of a shading contour.

7.3. Shaded contour

Contour shading is a common approach to visualize numerical data. Figs. 19 and 20 illustrate 3D shading contours in continuous and graded color mode, respectively. These figures represent a temperature distribution in an investment casting derived from a simulation of the process. In these figures, the control panels show the control facility available.

7.4. Weather map

The weather map is a new style of visualization for scalar fields, in which the scalar range of interest is specified. Only the elements, for which at least one of the nodal values falls in the specified range, are shown. Fig. 21 is an example of a weather map. It is obvious that the 3D version of the weather map is able to provide presentation of the physical processes ongoing inside the solid object.

8. Vector and tensor fields

8.1. Vector representation

In a discretized domain, a vector field \( \mathbf{V} = \{V_i\} \) can be visualized by using 3D arrows (icons). All these 3D arrows in the geometric space can be obtained by means of geometric transformation from an unit arrow as an arrow icon. Each arrow incorporates \( 4 \times 4 \) real matrices for homogeneous transformations. If \( \mathbf{p} \) is a 4-element row vector representing homogeneous coordinates of a point in the OOGL object, and \( \mathbf{A} \) is the \( 4 \times 4 \) matrix, then the transformed point is \( \mathbf{p}' = \mathbf{pA} \). The matrices applied to the unit arrow are associated with scaling, rotating and translating.

Fig. 22 shows a velocity field in a 2D mould filling problem [14], where the velocity field describes the filling process of liquid metal as it flows into a casting mould. Figs. 23 and 24 give quasi-3D representation, where the color shading is related to the temperature distribution while the depth of the metal in the mould reflects the completeness of the filling process.
Fig. 22: Velocity distribution in a 2D mould filling problem.
Fig. 23. Quasi 3D representation of a 2D mould filling problem.
Fig. 24. Camera view of the quasi 3D representation for the 2D mould filling problem.
8.2. Tensor representation

Because of the wealth of multivariate information in tensor fields, tensor visualization is a challenge. In this version of FEView, only second-order tensor fields are considered. Generally, a 3D second-order tensor field $T$ consists of a $3 \times 3$ array of scalar functions $\{ T_{ij} \} (i, j = 1, 2, 3)$ defined over a 3D domain. Independent visualization of these nine functions is possible, but has less meaning.

For a symmetric tensor field $U = \{ U_{ij} \} = \{ U_{ji} \}$, at every point, there are three real eigenvalues $\lambda^{(i)} (i = 1, 2, 3)$ as well as three real and orthogonal unit eigenvectors $e^{(i)} (i = 1, 2, 3)$. Therefore, the tensor field $U$ can be visualized by means of three orthogonal vector fields $v^{(i)} (i = 1, 2, 3)$, according to $v^{(i)} = \lambda^{(i)} e^{(i)}$. That is, visualizing $U$ is fully equivalent to visualizing the three vector fields $v^{(i)}$ simultaneously, since they include all the amplitude information and all the directional information.

It should be mentioned that the eigenvectors of an unsymmetric tensor field $T = \{ T_{ij} \}$ are generally complex and not orthogonal. Visualizing an unsymmetric tensor field is more difficult. However, an unsymmetric tensor can be decomposed into two components: a symmetric tensor and antisymmetric tensor as follows:

$$T_{ij} = \frac{1}{2} (T_{ij} + T_{ji}) + \frac{1}{2} (T_{ij} - T_{ji}).$$

![Fig. 25. Arrows to represent tensor fields in stress analyses.](image)
And in three-dimensional space, a second-order antisymmetric tensor is equivalent to a vector, since this tensor has only three independent components that form a vector known as the axial vector. Therefore, a straightforward way is to represent the unsymmetric tensor in terms of a symmetric tensor and a vector with different colors.

The tensor icon used here is a combination of three orthogonal arrows. It is also possible to visualizing symmetric tensor field using other tensor icons such as Haber's tensor glyphs or ellipsoids having the three vectors $v^{(i)}$ for principal axes [4]. Fig. 25 shows arrow icons representing a nodal tensor of stress, in which one is compression and the other two represent tension stress components.

8.3. Discussion

The vector and tensor icons explained above represent all the vector and tensor information at a given location, but their discrete nature does not reveal the underlying continuity of the data field. For this reason, efforts have been made to visualize these fields with streamlines. In particular, a generalization of the streamline, a hyperstreamline, has been designed to represent tensor fields, and it is reasonably impressive [15]. However, much more computing cost will be involved for the continuity of the data field. In general engineering applications, it is normally enough to represent the discrete vector and tensor fields by means of the corresponding icons.

9. Animation

An animation session provides an interactive means to visualize dynamically the finite element objects in a sequence. There are several simple animation controls such as single frame stepping, as well as bouncing (i.e. adjustable speed playing). Fig. 26 shows an example of an animation process. In this animation mode, FEView writes finite element objects in OOG file formats to Geomview, which displays each frame of the sequence in order.

Essentially, the finite element objects, which can be visualized dynamically, are classified into two types: a sequence of FE meshes and a sequence of results data. The numerical data files are associated with mesh files by specifying mesh reference numbers \texttt{fe.idmesh}. Every single mesh is contained in one mesh file, and the results data at a single time step is stored in one single data file. All these files are ordered by putting suffices, so that the file names are of the form \texttt{file0001}, \texttt{file0002}, \ldots, of which the suffices are incremental integers. When the user start up the animation session, the base file name \texttt{file} should be specified first, then the program lists all the files available corresponding to that name. During the animation process, the file, for which the information is being visualized, is highlighted in the file browser.

For visualization of numerical data, the animation is carried on according to a controlled sequence. The corresponding mesh files are picked up as the results data files are assigned with reference numbers \texttt{fe.idmesh}, which specify the addresses of the meshes. This scheme is useful, especially in adaptive finite element analyses, where several data files are related to one file containing one single mesh.
Fig. 26. An example of an animation process.
The first and most obvious way of animation is by "playing" the frames. Clicking on the play button will initiate playing through the frames in an appropriate sequence, starting with the first frame or whichever frame FEView stopped in. The speed at which frames are animated depends on the setting of speed, i.e. time steps. There are buttons "(Step" and "Step)", for which animation can be stepped through in backward and forward directions.

During the animation process, color scale for scalar fields, and arrow scales for vector and tensor fields can be set according to each frame, or they can be scaled globally for all frames to enable comparison of quantities throughout the sequence in a consistent manner.

10. Local analysis

10.1. Bi-directional communication

Generally, an external module can communicate with Geomview in both reading and writing directions. Apart from the communication in which the external module only writes to Geomview, that in the reverse direction is to be described herein. There are two types of communication that can go from Geomview to an external module: asynchronous and synchronous communications.

In asynchronous communication, Geomview sends expressions that are essentially echoes of GCL commands. The external module sends Geomview a command expressing interest in a certain command, and then every time Geomview executes that command, the module receives a copy of it and responds to it. In the other type of communication, the module sends a request to Geomview for some piece of information and waits for a response to come back before doing anything else.

FEView uses the OOGL lisp library to parse and act on the expressions that Geomview writes to the module's standard input, and the communication is in the asynchronous mode. FEView can receive user pick events, i.e. when the user clicks the right mouse button with the cursor over a geometry object in a Geomview camera window. When this happens Geomview generates an internal call to a procedure called pick; the arguments to the procedure give information about the pick, such as what object was picked, the coordinates of the picked point, and etc. If FEView has expressed interest in calls to pick, then whenever pick is called Geomview will echo the call to FEView's standard input. Then FEView can respond to the pick information.

When the local analysis mode is set, FEView initializes the lisp library by calling LInit( ). It then calls LDefun to tell the library about the pick procedure, which is defined with a call to the PICKFUNC macro. This is the function for handling pick events. Then it sends a bunch of setup commands to Geomview, which includes defining a handle called littlebox that stores the geometry of a little box. Next it sends the command

\texttt{(interest (pick world \\
\hspace{1cm} nil nil ***)

which tells Geomview to notify FEView when a pick event happens.

10.2. Picking and analysing

In the development of mesh generators and finite element analysis programs, it is useful to check local information in the geometry domain of interest. Also, this information is helpful for the user
Fig. 27. The layout of local analysis.
to understand the physical phenomena in the visualization stage. FEView has been designed to provide this functionality. The corresponding control panel is identified as Local Analysis. Fig. 27 illustrates a layout of this type of analysis.

On starting up the interaction in local analysis mode, the button Locate is to be pushed first. Whenever the user picks on a point, FEView identifies the position by drawing a little box at the spot. Usually the box is yellow. If a nodal point of the finite element mesh is picked, then the box is colored magenta, and the node coordinates, node number and the associated numerical results are printed in the panel. If a point on an edge of the mesh is selected, then FEView will also highlight the edge by drawing cyan boxes at its endpoints and drawing a yellow line along the edge, meanwhile, the two node numbers of the endpoints are shown in the control panel. If a face of a mesh is chosen, then the face number and the associated node numbers are display in the corresponding position of the control panel.

11. Graphical user interface

11.1. Forms library

The graphical user interface in the visualization facility is built on the top of the Forms Library, a graphical user interface toolkit for Silicon Graphics workstations [16]. Constructing user interfaces for programs is normally a time consuming process, however the interfaces are very important to help the user work with the program in an easy and pleasant way. The Forms Library is handy and relatively simple, graphically good looking and easily extendible. This Forms Library consists of a large number of C-routines, which can be used both in C and in C++ programs.

The main notion of the Forms Library is that of a form. A form is a window in which different objects are placed. Such a form is displayed and the user can interact with the different objects on the form to specify his/her interests. Many different classes of objects exist, such as buttons that the user can push with the mouse, sliders with which the user can indicate a particular setting, input fields in which the user can provide textual input, menus from which the user can make choices, browsers in which the user can scroll through large amounts of text, and so on. Whenever the user changes the state of a particular object on one of the forms displayed the application program will be notified and can take action accordingly. The application program has a large amount of control over how objects are to be drawn on the forms. It can set colors, shapes, text styles, text sizes, text colors and etc.

11.2. Form designer

The Form Designer, coming with the Forms Library, makes the interface programming even easier. This program allows the user to design dialogue forms interactively and to generate the corresponding C-code. The user can simply choose the objects from a list to place on and to draw on a form. Then the user can set attributes, change sizes, positions of the objects etc., using only the mouse or the function keys. The piece of code generated by Form Designer will contain one procedure create_the_forms() which the application program needs to call to generate all the
forms designed. The *Form Designer* also allows the user to give names to objects for reference in the application program and allows binding of call-back routines to objects.

11.3. Interactions

The *Form Designer* only helps the user in designing the layout of the forms, it does not allow the user to specify the actions that have to be taken, for instance, when a button is pushed. The user can indicate the callback routine to call, but the application program has to supply this callback routine.

The application program can bind a call-back routine to any object. Whenever `fl_do_forms()` or `fl_check_forms()` return the object, the call-back routine is called instead. If no call-back routine is bound to the object then it is returned immediately. To bind a call-back routine to an object, the procedure to be adopted is

```c
void fl_set_call_back(FL_OBJECT *obj, void (*callback)( ), long argument)
```

where `callback` is the call-back routine, and `argument` is an argument that is passed to the call-back routine such that it can take different actions for different objects. The call-back routine should have the form

```c
void callback(FL_OBJECT *obj, long argument)
```

where `obj` is the object that causes the call-back routine to be called.

12. Conclusions

In the present work, the Geomview program has been used as a display engine, to which the code developed, FEView acts as an external module. The two-way interaction provided makes both Geomview and FEView work simultaneously. The *Form Designer* has been used and is shown to be a good toolkit to build a graphical user interface.

FEView deals with finite element meshes as collections of faces with edges, wire frames, or point clouds, and uses color shading and field icons (such as arrows) to visualize the associated numerical results. Numerical results of a two-dimensional problem can be represented with three-dimensional effects by using values of scalar fields as the third geometry coordinates. FEView provides animation control over single frame stepping and adjustable speed playing. It has been equipped with geometry operation functionality, in which a particular part of an object can be identified by specifying material indices, element numbers, and cutting boxes. In local analysis mode, FEView is able to provide local information about finite element objects by picking up the position of interest in the Camera window.

Acknowledgements

The authors wish to thank the EPSRC for funding the research work carried out under grant Number GR/G/35107. We would like to thank the creators of Geomview/OOGL for permission
to use their software, and Prof. Mark Overmars for the use of the Forms Library. The authors are grateful to P.I. Castings Ltd., UK, for their industrial input via the investment casting example.

References