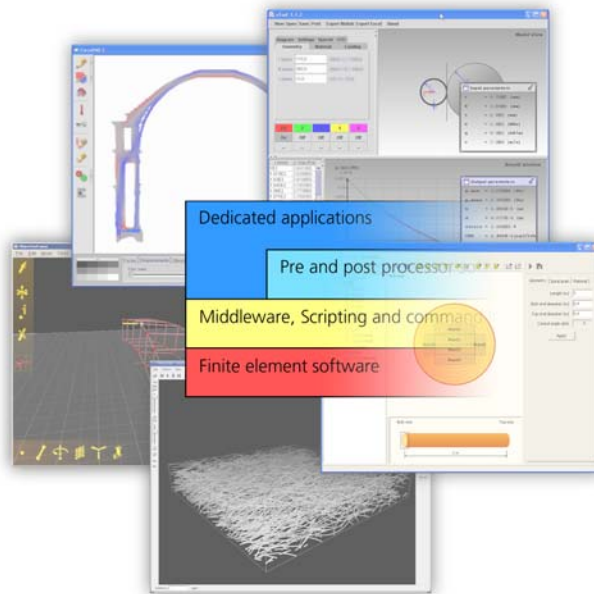# TECHNIQUES FOR DISTRIBUTED ACCESS AND VISUALISATION IN COMPUTATIONAL MECHANICS

JONAS LINDEMANN

Structural Mechanics

Doctoral Thesis

Structural Mechanics

TECHNIQUES FOR DISTRIBUTED ACCESS AND VISUALISATION IN COMPUTATIONAL MECHANICS

Doctoral Thesis by

JONAS LINDEMANN

Preface

The work presented in this PhD thesis was carried out at the Division of Structural Mechanics, Lund University.

I would like to express my gratitude to my supervisors, Professor Ola Dahlblom and Professor Göran Sandberg for their support and for the discussions we have had regarding the topics in this work. I would also like to thank Karl-Gunnar Olsson for contributing with the fundamental ideas and discussions of the design and educational aspects of using finite element software in an educational setting, and Mr Bo Zadig for helping me put it all together.

There are of course many other people, colleagues of mine at Structural Mechanics included, who in some way have contributed to the work.

Finally, I am grateful for the opportunity of using the computer resources available through Lunarc, the center for scientific and technical computing, at Lund University

Lund, September 18, 2003

Jonas Lindemann

Contents

Part I

Introduction and overview

Chapter 1

Introduction

Computational mechanics software is widely used in many disciplines today. For the most part, it is used by people with a good understanding of the methods involved. The way in which finite element applications has been used have not changed much since they were first conceived. The normal workflow is shown in Figure 1.1. A pre-processor is used to define the geometry, loads and boundary conditions. The pre-processor then generates a mesh, which in turn is used for generating an input file for the finite element solver. The solver writes the results to an output file, which is read into a post-processor for
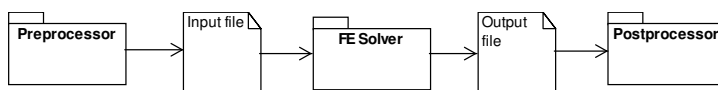
Figure 1.1: Normal workflow of a finite element application

visualisation. The design of these applications imposes certain restrictions on the use of computational mechanics software, concerning both its usability and its integration with other software.

The work presented here aims at extending the context of computational mechanics software into such areas as computer science, scientific visualisation and human-computer interaction. **Papers I**, **II** and **III** deal with methods for accessing finite element software by use of distributed technologies. **Paper IV** presents a method of visualising computational results in real-time. **Papers V** and **VI** examine the usability of the graphical user interface, particularly in an educational setting.

Chapter 2

Overview

2.1 Different levels of interfaces

Finite element software often has many user and developer interfaces. To make it easier to understand how users and developers can interact with finite element software, interfaces of four different levels are defined, see Figure 2.1
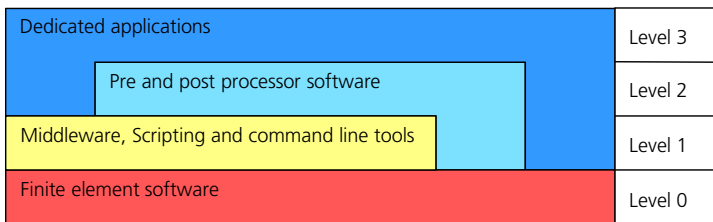
Figure 2.1: The different levels of interfaces to finite element software

Each level also has two interface categories, the user interface and the developer interface. The finite element software with its various software components, is at level 0. The user interface to the finite element software being defined here by input and output files. The developer interface (if one exists) is defined by header files to library routines for input and output.

At level 1, the user interface to the software is made less complex and easy to use by the provided scripting and command-line tools. The user is given greater control and can use the software in a more flexible way. The developer interface at this level can also use scripting and command-line tools. Direct access to the software is provided by middleware software enabling a higher-level interface and a component-based architecture. Special types of middleware can also provide distributed access to the finite element software.

At level 2, the user and developer interfaces are defined by pre- and post-processor software, which makes the generation of input data and the analysis of the output data

more effective. The pre-processor generates the input data for the finite element software, often using simple geometric shapes to define the object being modeled. This frees the user from the task of specifying each element in the finite element mesh. The post-processor software analyses the output of the finite element software and presents the results in terms of 2D and 3D plots.

At level 3, the user interface is defined as a dedicated application which solves a specific problem, the details of the finite element software being hidden from users.

Applications written at a specific level often use interfaces located at several levels. A dedicated application at level 3 can be implemented to use finite element software directly at level 0 or can access the finite element software by use of middleware or pre- and post-processing software at level 1 and 2.

2.2 Distributed technologies and middleware

Classical distributed technologies are often based on a client/server architecture that allows clients to communicate with servers directly. This approach works well when protocols are well-proven and well-established, but is often difficult to implement if the systems become more complex. When changes and features are added on the server side, the client often needs to be updated. To solve this, an additional layer called a middle-tier (level 1 in Figure2.2) is placed between the client and the server. CORBA [1], DCOM [2], Java RMI [3], PHP [4] and .NET [5] for example, are technologies that have been developed to create this middle layer. The main idea is that a set of interfaces is defined in the middleware software describing the services provided. Clients developed against these interfaces communicate with the middleware software instead of the real server system. This allows the server side to be updated and reconfigured without any of the clients being affected by this. Once an interface has been defined, it can not be changed. Functions can be added by new interfaces being added to the middle layer. There are also other benefits of using middleware technologies. Some technologies such as CORBA, DCOM and .NET are language-neutral. Interfaces are often defined by use of a special language. From the definition determined in this way, client and server code can be generated for any language that is supported.
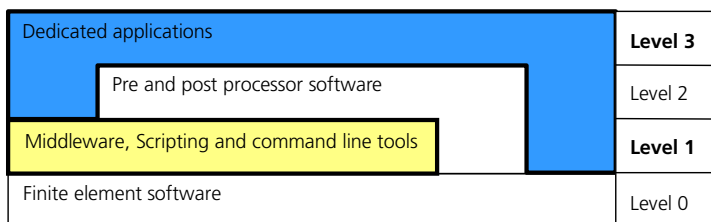

Figure 2.2: Interface levels of distributed technologies and applications

Finite element software today still uses text files to a large extent for communication with pre- and post-processors. Most finite element software uses a special proprietary-standard file format. Pre- and post-processor developers often need to support a variety of

different finite element codes. This requires the development of a large number of export filters. However, these need to be updated whenever updating of the input and output file format occurs. Use of text files for communication has other drawbacks as well, such as ineffective storage schemes. Standardisation efforts often fail because it is too slow to support many of the features introduced in new versions of finite element softwares. A solution to this is to use a middle layer between the pre-processor and the finite element code, just as implemented in modern client/server applications. The middle layer consists then of a set of general interfaces describing the input and output of the finite element software. In a perfect world, developers providing finite element software would create a set of general interfaces for defining such matters as the geometry and properties of the elements and the results output. Each developer could then add specialised interfaces for specific feature of their own software. This would make it easier for pre- and post-processor developers to support a wide range of finite element software. Interfaces to different packages can be generated in any language automatically from the definitions the finite element developers provide.

One advantage of using middleware technology is location transparency. This means that a client using a specific interface does not need to know whether the resources in question are situated locally or remotely. This in turn allows the computational resources to be used and configured in a variety of ways without users having to configure their client applications in any special way.

The new trend which is emerging in distributed computing today is that of GRID computing. GRID computing connects computational resources such as clusters with each other to form a "Meta cluster". Both clusters and grid resources are oriented to command-line interactions. Although users need to be familiar with a wide variety of tools and commands in order to use the resources available effectively, this has not been a problem up to now because most current users of clusters and GRIDs are familiar with these tools. To gain acceptance for these technologies by a wider user group, however, the interface to these resources needs to be improved. Middleware technologies such as CORBA [1], PHP [4], .NET [5] and Java RMI [3] can be the key to providing improved interfaces to such resources. Figure 2.3 shows an example of a Java-based engineering tool able to take advantage of distributed resources by executing finite element calculations remotely. Figure 2.4 shows an effort under way at Lunarc [1] to create a more user-friendly interface to cluster and GRID resources there by providing much of the functionality of the cluster in a web-based form.

Paper I describes a three-tier finite element application using Microsoft's Distributed Component Object Model (DCOM). **Paper II** provides a more in-depth look at the use of middleware technologies in computational science and in finite element software. The performance of CORBA for the transfer of large amounts of data is studied as well. The paper also includes a complete description of a CORBA-based finite element implementation. **Paper III** describes a new finite element application with a plug-in interface for finite element solvers. Plug-ins will be made available for both local and remote execution by use of CORBA or other distributed technologies.

---

[1]Lunarc is the centre for scientific and technical computing at Lund University
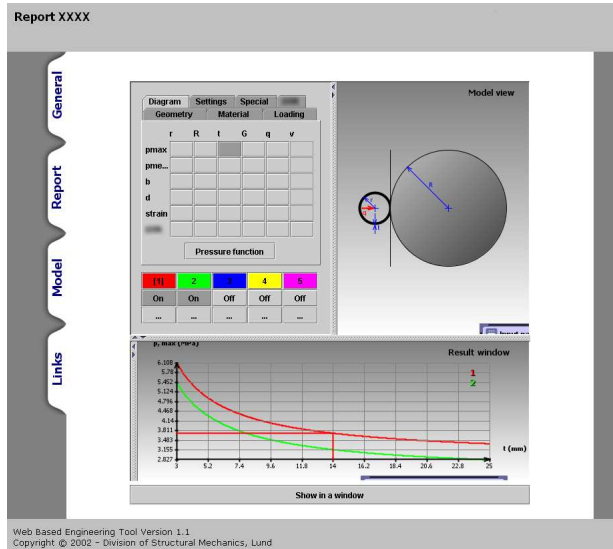
Figure 2.3: eTool, a web-based engineering tool
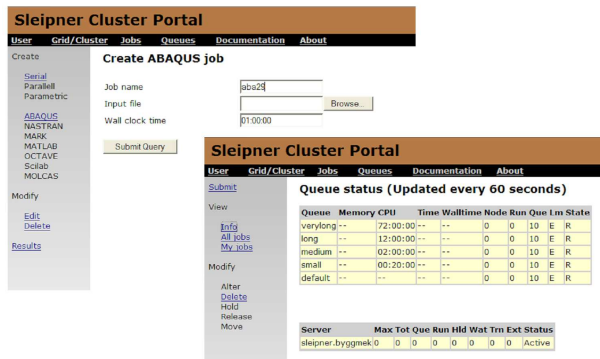
Figure 2.4: Web-based interface to cluster and GRID resources

2.3 Visualisation

In many situations, normal post-processors cannot be used for visualising results. Susanne Heyden [6] has developed a software for simulating deformation in three-dimensional fibre structures. The code involved produces result files which cannot be readily used in conjunction with commercially available post-processors. The complexity of the model, evident in Figure 2.6, requires special methods for efficient visualisation. The methods are implemented in the C++ OpenGL [7] application FibreScope. During development of the FibreScope application, different methods for visualising fibre structure were investigated. In the one method, a circular cross-section is swept along the fibre spine,

creating an extruded fibre shape, see the image at the left in Figure 2.5. Although this method provides a good representation of the fibre geometry, a large number of triangles are generated. A second method, in which the geometry of the fibre is reduced to a single band consisting of two triangles per fibre segment, was developed for visualising larger fibre networks. A texture is also applied to the band for representing the fibre structure. Visibility issues are dealt with by turning the band toward the user at each vertex point on the fibre. The result is a visualisation with significantly fewer triangles. An example of this method is shown in the image at the right in Figure 2.5.
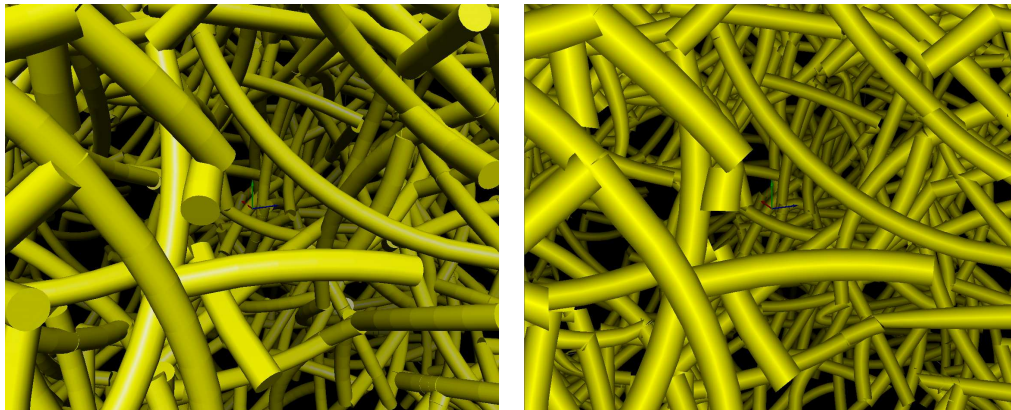
Figure 2.5: Fibre visualisation using the extrusion based method (left) and the band method (right).

The FibreScope application was also used in a recent project [8] to help evaluate methods for generating random fibre networks of differing properties. With use of the application large networks could be rotated and viewed on the screen in real-time, revealing quickly whether the generation algorithms were working. The generation algorithms will probably be integrated into the FibreScope application, allowing the latter to be used as a pre-processor as well. Figure 2.6 shows FibreScope's main user interface.

FibreScope is implemented in C++ using the FLTK [12] library for the 2D user interface parts and the Interactive Visualisation Framework Ivf++ [13] for the visualisation parts. Ivf++ is a C++ Scene graph library using OpenGL for rendering. Through use of FLTK and Ivf++, FibreScope can be utilised on any platform, such as Linux, SGI/IRIX or Microsoft Windows, having an OpenGL implementation.

Paper IV describes the various visualisation methods used in FibreScope, comparing them in terms of performance.
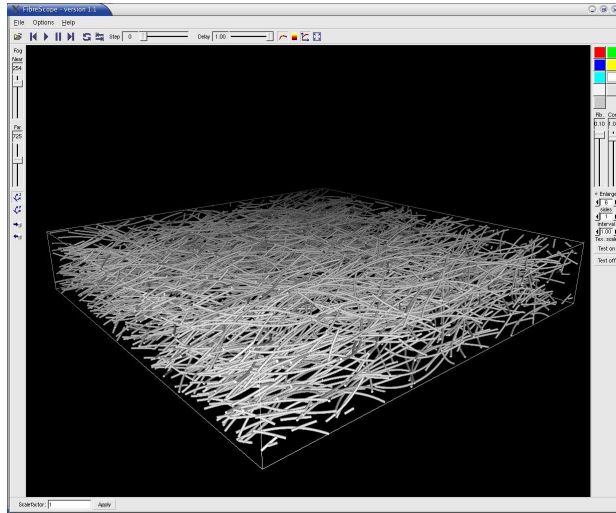
Figure 2.6: The FibreScope application

2.4   Graphical user interfaces

Finite element software can be used in a wide variety of areas, due to its generality. One problem, however, is that finite element software packages often require a thorough understanding of the inner workings of the software (level 2). To allow the benefits of finite element software to be taken advantage of adequately, the user interface should be implemented on a higher level, such at level 3 in Figure 2.1. The interface should also be designed with the target-user group in mind.
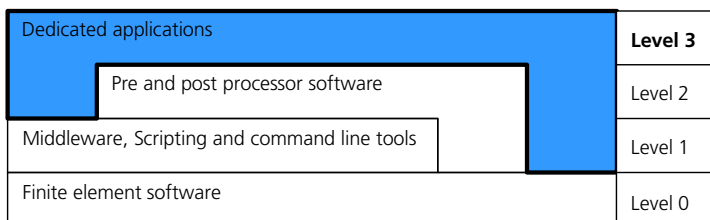
Figure 2.7: Interface levels for graphical user interfaces

Teaching mechanics to engineering and design students involves providing students with the ability to go on from investigating and understanding properties involved to becoming actively engaged in designing, articulation and expression, going from inner qualities to outer contours. Tools particularly useful in this process are ones involving simple sketching and enabling rapid responses to be made in investigating different mechanical properties such as those of contour deformation, forces and force fields. Sections 2.4.1 and 2.4.2 describe certain tools developed for this purpose.

2.4.1 ForcePAD

In design and in design education, much emphasis is placed on the concept of sketching. A design is never accepted automatically, but is iterated over time until a satisfactory solution is found, sketching being used extensively in this process. ForcePAD [9] was developed as a tool both for enhancing the users understanding of mechanical concepts and for use in the sketching process, particularly in an educational setting.

The ForcePAD application shown in Figure 2.8 employs metaphor similar to the metaphors found in such image editing applications as Adobe Photoshop [10] and Jasc PaintShop Pro [11]. These applications are generally very intuitive and direct, using pens,
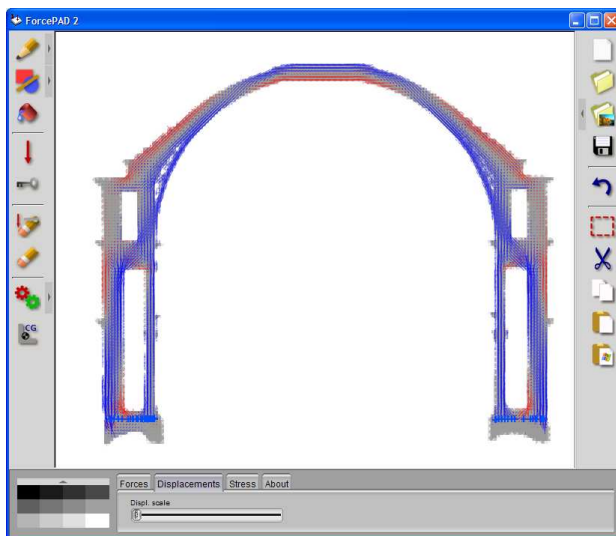
Figure 2.8: The ForcePAD application

brushes and colour palettes as tools for drawing. The ForcePAD user interface is based on the same principles. The main difference here is that painting is done not in colour but with use of a grey scale in which white represents zero stiffness and black maximal stiffness. Use of the painting metaphor also provides other benefits. ForcePAD is able to import pixel-based images from files or from the Windows clipboard. Imported images are automatically converted to grey scale images. Use of this approach makes it easy for a design student to take a sketch, scan it and then paste it into ForcePAD, where displacements and stresses can be analysed. Figure 2.9 shows an example of this process. In the example, ForcePAD is used to import a scanned image, edit and then analyse a sketch of the Pantheon in Rome. The complete analysis can be carried out in a matter of minutes.

The ForcePAD application also supports the study of mass, centre of gravity and equilibrium by use of a special version called ForcePAD/R. Figure 2.10 shows the results of an assignment in which students photographed objects which they then analysed in terms of centre of gravity and equilibrium.
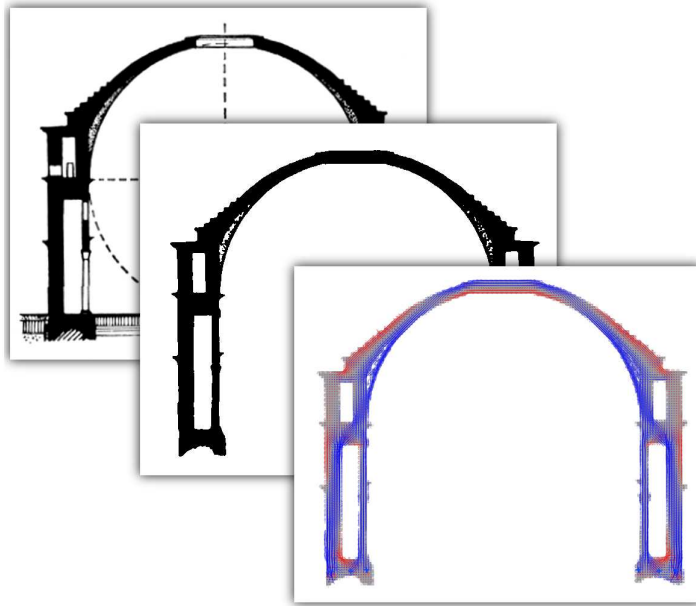
Figure 2.9: ForcePAD example involving scanned images
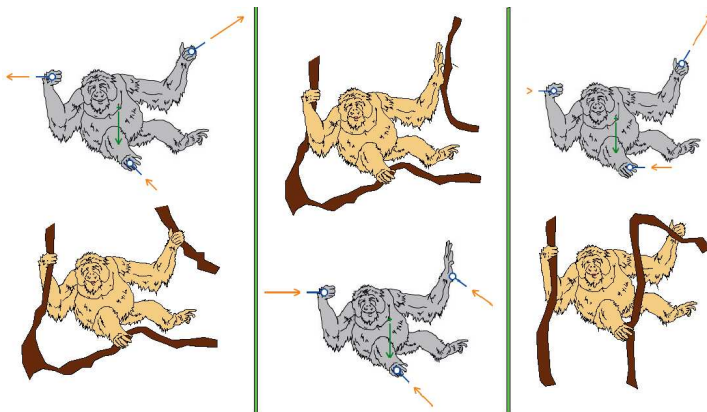
Figure 2.10: This brief series of pictures shows how a student coupled mechanical entities with expression by use of the ForcePAD/R [9] software. The direction of a branch on the right side in (a) and (b), and thus the support conditions as well, are manifested in the positioning of the hand, which is closed (a) and open in (b). Note the changes in the support load that occurs.

ForcePAD is implemented in C++ by use of a set of platform independent libraries. The finite element code is implemented using the NEWMAT09 [14] matrix and solver

library. The graphical user interface is implemented in the Fast Light Toolkit (FLTK) [12]. This is a user-interface library available for Linux, Mac OS X, Windows and most versions of UNIX. The library is very efficient and produces highly responsive applications on any platform. Another advantage of FLTK is that of the graphical user interface designer FLUID, which comes with the library. FLUID provides almost the same level of rapid application development as Borland Delphi [15] and Microsoft Visual Basic [16], but its also providing platform independence. Drawing and visualisation are implemented with the use of the OpenGL [7] graphics library. OpenGL is generally regarded as a 3D graphics toolkit, although it also has an effective 2D rasterisation interface enabling hardware-accelerated drawing to be performed rapidly. With the use of this approach, ForcePAD allows rapid sketching to be done and the finite element meshes to be updated continually, enhancing the directness of the actions carried out.

Paper V describes the design and implementation of ForcePAD and also presents an educational case study.

2.4.2   ObjectiveFrame

ObjectiveFrame, see Figure 2.11, was conceived for developing new ideas and principals for user interaction in 3D finite element software. ObjectiveFrame [17] is a 3D frame-analysis tool implemented by use of OpenGL. The user interface for it is designed so as to resemble the way modeling is done in a shop, thus creating a virtual shop. To accomplish this, interaction with the model needs to be immediate and the representation sufficiently clear, for a user to feel immersed in the model. This is done by providing ObjectiveFrame with a fully lit and shaded 3D model, together with direct feedback concerning interaction with the objects in question.
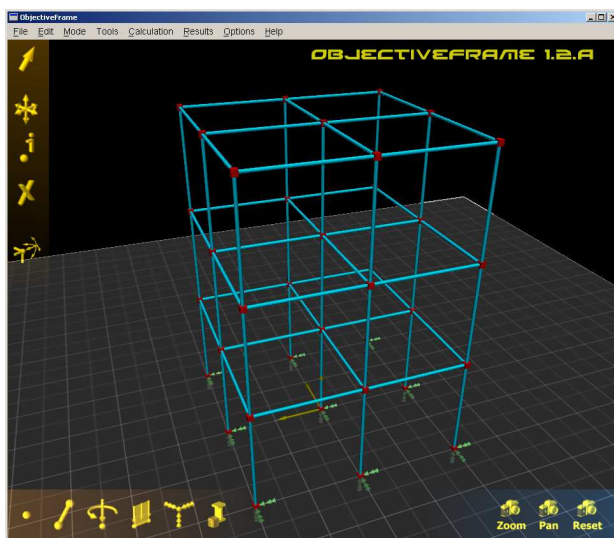

Figure 2.11: ObjectiveFrame

13

One of the most important features of ObjectiveFrame is its ability to visualise the response of a structure subjected to a user-controlled load in real-time. This enables users to "feel" the degree of stiffness in different directions in a structure. An example of this is shown in Figure 2.12. The real-time features of ObjectiveFrame have also been
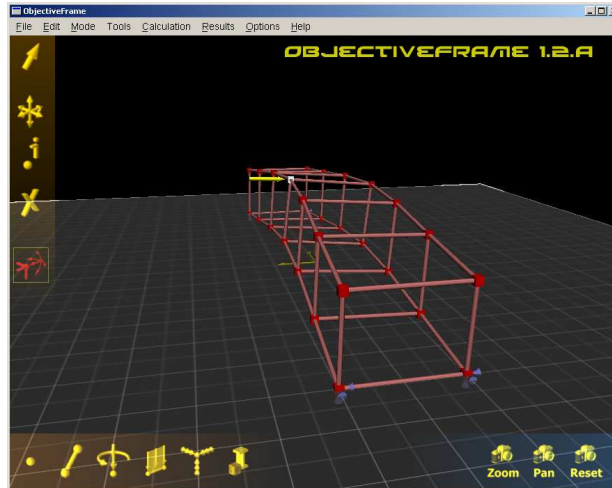
Figure 2.12: Real-time updating of a structure in ObjectiveFrame

exploited in a course in architecture [2] in which students investigate different techniques for the construction and stiffening of high structures, experimenting with the techniques in question and exploring the effects these have by "feeling" the mechanical properties of the structure.

The ObjectiveFrame application is implemented in C++, using a set of platform-independent libraries. The finite element code is implemented by the use of the NEW-MAT09 [14] matrix and solver library. The graphical user interface is implemented in the Fast Light Toolkit (FLTK) [12]. Rendering of 3D graphics is involves the use of the Inter-active Visualisation Framework Ivf++ [13], a C++ scene-graph library, using OpenGL for rendering. The ObjectiveFrame application is taken up in **Paper VI**.

The initial usability testing of the ideas and principals in ObjectiveFrame was done in the different courses in which it was utilised. This has led to continued developments of it. However, even though the usability of an application can be tested rather well in a classroom setting, not all the answers needed can be obtained in this way. To improve the user interface of ObjectiveFrame still more, an initial usability study was conducted early in 2003, see **Paper A.1**. This study involved analysis of the existing user interface, and a user test of a new interface design. This new design introduced a more direct approach to selection, creating of loads and treatment of the boundary conditions. It also introduced a new and improved method for handling the cursor. Some of the improvements are shown in Figure 2.13. The improvements achieved on the basis of this usability study will be included in the next major release of ObjectiveFrame.

---

[2]A course in high structures given at the School of Architecture at Chalmers University of Technology
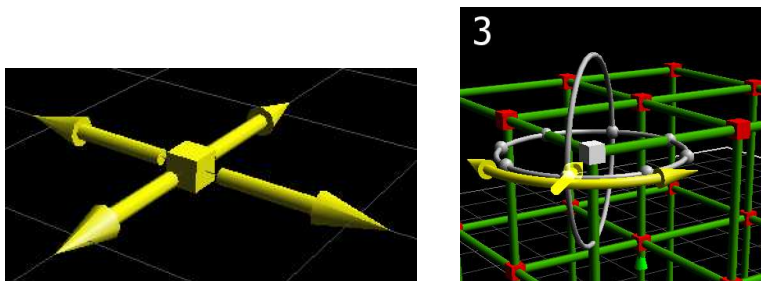
Figure 2.13: New cursor and load handling in an ObjectiveFrame prototype

2.5 Visualisation framework

Hardware-accelerated 3D rendering is standard on most platforms today. The most common way of accessing 3D hardware is by means of an API (Application Programmers Interface). A common API is OpenGL which is platform independent and available on several hardware platforms. Programming a visualisation application in OpenGL can be a complicated task, since OpenGL is a low-level API, most advanced functions such as view transformation and rendering of advanced geometry need to be implemented by the developer. Performing a given task often involves using several OpenGL calls. A higher-level library such as Open Inventor, OpenGL Optimizer or OpenGL Performer is frequently employed to make 3D rendering by means of OpenGL easier. An object-oriented approach is often used to implement libraries of this type in C++. Major disadvantages of such libraries are that they are often designed for a specific task and that they tend to be large, complex, and difficult to extend.

The Interactive Visualisation Framework, Ivf++, was developed as an object-oriented layer to be placed on top of OpenGL. This library was also the basis for the FibreScope and ObjectiveFrame applications. The library implements a scene-graph, as well as a framework for basic user-interfaces and special widgets for interfacing with user interface toolkits. Ivf++ is an open-source library available for downloading at Sourceforge [25] [3]. Some of the Ivf++ features are the following:

- Modular library design. Only the parts needed are used.
- Built to be extended.
- Platform independent, compiling on Windows, Linux or SGI/Irix.
- A scene-graph with culling support.
- A reference counting system with smart pointers.
- Texturing supported by image loaders JPEG, PNG, TIFF, TGA or SGI rgb-files.
- 3D file format support for DXF, AC3d models and polyfiles.
- A user-interface library for creating simple OpenGL applications in FLTK, MFC and native WIN32.

---

[3]Ivf++ 0.6.0 was released as an open source library under the LGPL license [18] in February 2000 . The current version is 0.9, which since its creation has been downloaded over 12000 times (September 18, 2003)

- 3D user interface controls.
- Stereo support.
- Complete class documentation.
- A User guide.

Paper A.2 describes the major ideas and guiding principles of the Ivf++ library.

Chapter 3

Concluding remarks

This work presented here concern ideas and methods on how the context of computational mechanics can be improved for more efficient utilisation and extended into areas previously not familiar with or capable of using such tools. Three main areas were studied:

- Providing efficient and transparent access to finite element applications.

- Developing methods for the visualisation of complex phenomena.

- Improving the usability of the finite element method, partly in an educational context.

An appoach to providing more efficient and transparent access to finite element applications and to other computational mechanics software and libraries, based on the CORBA specification is introduced and its usefulness explored.

Enhancing the understanding of complex phenomena is highly important. Many computational codes produce large amounts of simulation data that need to be analysed and evaluated. By creating tools that can visualise these simulations in real-time, understanding of physical phenomena involved is enhanced. A special method for the visualisation of large fibre networks, one that increases the size of the networks that can be visualised in real-time, is also introduced.

Improving the usability of finite element software is important if it is to be employed effectively in an educational setting. Normal computational mechanics codes are often based on a very flexible hierarchical model, which is an obstacle for users unfamiliar with it. Two approaches which appear more practicable in this respect are introduced here. The one approach involves use of a new direct-image-based metaphor for creating a tool that can be thought to ultimately be useful in facilitating creative processes in the work of engineers, designers and architects. A second approach aims at improving interactivity and the understanding of mechanical concepts in a 3D frame application using real-time feedback of both the interface and the resulting deflection of the structure.

3.1   Transparent access to finite element software

The design of computational mechanics software has not changed fundamentally over the years. The basic computational code is designed as a normal console application, files or a relational database being used for communication and storage purposes.

In computer science, many new technologies have emerged which facilitate the more flexible use of applications. Many applications today have an embedded script language, such as Python [19], Visual Basic for Applications (VBA) [20], Tcl/Tk [21] or Ruby [22]. These script languages enable users to readily extend and utilise applications in ways that would otherwise have required their being recompiled. Distributed technologies such as .NET [5], CORBA [1] and Java RMI [3] provide a middle layer for distributing resources over the internet. The use of computational mechanics software can be made more flexible and efficient by use of such technologies.

User interface codes are often implemented in Java, C or C++. Interfacing computational codes based on these languages often requires special interface layers, the development of which can be time-consuming. Through use of CORBA and the interface definition language (IDL) for describing the functionality of computational mechanics codes, many of these problems can be solved. Using IDL, functionality of a computational code can be defined in a language neutral way allowing the code for interfacing with IDL-specified objects and functions to be generated in any desired language automatically. The original computational code can still be kept in the original implementation language, providing stability and maintainability.

Another way of accessing a computational code is by providing an interface to a script-language. Through using CORBA when developing computational codes, interfacing with script-languages becomes an automatic process. Some CORBA implementations, such as fnorb [23] and omniORB [24] support the generation of interfaces to script languages directly from the IDL-definitions involved.

Computational mechanics codes implemented with use of a CORBA interface also take advantage automatically of the distributed features of CORBA. Location transparency is implicit in the CORBA specification. A client application accessing CORBA objects or functions does not need to be implemented in any special way for calling remote or local objects. This enables computational mechanics codes to be placed on powerful computational resources allowing clients located either remotely or locally to access them. Client applications involving either aweb-based client or a stand-alone client can access the resources available, providing for efficient use of the computational resources.

The implementation and performance characteristics of using CORBA and other distributed techniques in computational mechanics software are taken up in **Papers I**, **II** and **III**.

3.2   Visualisation of complex phenomena

The visualisation of complex phenomena in results provided by computational mechanics codes is important for the evaluation and understanding of physical phenomena. For the

analyses carried out to be efficient, and understanding of phenomena that is provided to be adequate it is also important that the visualisation obtained can be interacted with the in real-time. This enables results to be animated, providing for a better understanding of time dependant effects.

Efficient real-time rendering also facilitates computational steering. The results at each time step of a simulation being visualised allows the user to determine quickly whether or not the simulation is errorneous, allowing the simulation to be terminated if appropriate or the parameters to be changed in the course of the simulation, reducing the time required for analysis.

The large amount of result data that computational software produces, can be difficult to analyse and to evaluate if non-conventional geometries and enteties are employed. At the same time standard post-processors are often designed for standard element types only, having difficulties in dealing with non-standard elements. Visualising the behaviour of the thousands of fibres included in a fibre network simulation [6] often requires advanced 3D graphics hardware. Taking advantage of techniques developed in the field of scientific visualisation, such as billboarding, impostors and texturing, can improve performance and reduce the hardware requirements considerably.

A special textured billboard method was developed to increase the number of fibres that can be visualised in real-time. This method involves a line being swept along the fibre spine, reducing the triangles to two per fibre segment. Visibility issues are solved by orienting the band toward the user at each spine vertex. Due to the band fibre being flat, however, the fibres do not look round unless special measures are taken. These involve applying a special gradient texture to the band.

Paper IV describes the method implemented here and the post-processor software FibreScope developed for this method.

3.3 Usability and educational aspects of finite element software

Computational mechanics software is often designed to be very general, supporting several types of elements and differing geometries. The software typically employs a hierarchical description of the problem to be studied, see COSMOS [26], MSC/Patran [27], ABAQUS/CAE [28]. If the user is familiar with the conceptual model involved, hierarchical models can be both efficient and flexible, but if the user is not, the complexity of such models can be difficult to handle, see Shneiderman p.68 [29].

The overall usability of computational mechanics tools needs to be improved if these are to be used effectively in a broad context, such as in an educational setting or with divergent groups of users. The demands on usability placed on conventional computational software in an educational setting is often greater than that placed on it in an engineering setting. Students unfamiliar with finite element method is scarcely able to make adequate use of an advanced finite element package.

In a problem-based learning environment, applications need to support both experimentation and an iterative design process, creating a virtual workbench. The major

methods of implementing this in computational mechanics software are to make applications more direct and to provide rich feedback. To accomplish this, the principles used in the design of user interfaces in computational mechanics software need to be changed.

One approach suggested in the work reported here is to replace the classical hierarchical modelling approach by an image-based modelling metaphor, such as found in Adobe Photoshop [10] for example. Image-editing applications are often very direct and easy to use, being based on the direct-manipulation concept[1], creating structures as easy as selecting a brush and moving the pointer. The user can also see the results immediately. The ForcePAD [9] application described in **Paper V** implements the suggested image-based modeling metaphor in a 2D finite element application. Instead of drawing with color as in an image editor, the user draws with a grey scale, white representing no stiffness at all and black maximum stiffness. The user is able to quickly create and solve finite element models without having to spend time on modeling the geometry. The ForcePAD software has successfully been used in teaching students in engineering, architecture and industrial design.

In 3D finite element modeling, problems of usability are even more problematic. Many finite element pre-processors simply extend the metaphors found in mechanics textbooks to 3D conditions. This approach has two main drawbacks. One is that mechanics textbooks often concern 2D problems, where the metaphors in question work satisfactory. Extending these metaphors to 3D conditions is not always intuitive. A second drawback is that the symbols found in textbooks often require a thorough understanding of the underlying principles. To make 3D finite element modeling more accessible to a larger user group, the user interface needs to be improved. The ObjectiveFrame application described in **Paper VI**, was developed to study possible improvements that can be made in existing 3D finite element packages. ObjectiveFrame implements a user interface that responds directly to the user's inputs, any changes in the dimensions involved or rotation of a beam being instantly displayed. ObjectiveFrame also takes real-time feedback one step further. Users are able to "feel" the stiffness of a structure by placing a load on the structure involved the displacements that occur being updated and visible immediately in the 3D view. A small usability study concerned with a new version of ObjectiveFrame was carried out, a version which involved use of 3D widgets allowing load and boundary condition placement to be determined by use of a direct approach, see **Paper A.1**.

---

[1]A software using direct manipulation should possess the following properties: visibility of the objects of interest, rapidity and reversibility, incremental action, and replacement of a complex command language syntax by direct manipulation of the object of interest [30]

Bibliography

[1] Object Management Group Inc., http://www.omg.org, 2000

[2] Microsoft Corporation, DCOM Technical Overview, 1996

[3] Sun Microsystems Inc., JavaTM Remote Method Invocation, http://java.sun.com/-j2se/1.3/docs/guide/rmi/index.html, 2003

[4] PHP - PHP Hypertext Preprocessor, http://www.php.net, 2003

[5] Microsoft .NET Framework, http://msdn.microsoft.com/netframework, 2003

[6] Heyden S, A 3D Network Model for Evaluation of Mechanical Properties of Cellulose Fibre Fluff, Report TVSM-1011, Division of Structural Mechanics, Lund University, 2000

[7] OpenGL, http://www.opengl.org, 2003

[8] Edlind N, Modelling and Visualization of the Geometry of Fibre Materials, Report TVSM-5117, Division of Structural Mechanics, Lund University, 2003

[9] ForcePAD, http://www.byggmek.lth.se/resources/forcepad/forcepad.htm, 2003

[10] Adobe Photoshop 7, http://www.adobe.com/products/photoshop/main.html, 2003

[11] Jasc Paint Shop Pro 7, http://www.jasc.com, 2003

[12] B. Spitzak, Fast Light Toolkit FLTK, http://www.fltk.org, 2003

[13] Interactive Visualisation Framework - Ivf++, http://www.gorkon.byggmek.lth.se/ivfweb, 2003

[14] Newmat C++ matrix library, http://www.robertnz.net/nm_intro.htm, 2003

[15] Borland Delphi, http://www.borland.com, 2003

[16] Microsoft Visual Basic, http://msdn.microsoft.com/vbasic, 2003

[17] ObjectiveFrame http://www.byggmek.lth.se/resources/objectiveframe/objectiveframe.htm, 2003

[18] The GNU Lesser General Public License, http://www.fsf.org/licenses/licenses.html-#LGPL, 2003

[19] Python, http://www.python.org, 2003

[20] Microsoft Visual Basic for Applications, http://msdn.microsoft.com/vba, 2003

[21] Tcl Developers site, http://www.tcl.tk, 2003

[22] Ruby: Programmers' Best Friend, http://www.ruby-lang.org/en, 2003

[23] The pure Python CORBA ORB, http://www.fnorb.org, 2003

[24] The omniORBpy version 2 User's Guide, http://omniorb.sourceforge.net/-omnipy2/omniORBpy, 2003

[25] Sourceforge - Breaking Down The Barriers to Open Source Development, http://www.sourceforge.net, 2003

[26] COSMOSWorks, http://www.solidworks.com/pages/products/cosmos/cosmosworks-.html, 2003

[27] MSC.Patran,   http://www.mscsoftware.com/products/products_detail.cfm?PI=6, 2003

[28] ABAQUS Inc., http://www.abaqus.com, 2003

[29] B. Shneiderman, Designing the user interface : strategies for effective human-computer interaction, Third edition, Addison-Wesley, 1998

[30] J. Preece et al, Human-Computer Interaction, Addison-Wesley, 1994

Part II

Appended Papers

Paper I

An Approach For Distribution Of Resources In Structural Analysis Software

European Conference on Computational Mechanics - ECCM 99

AN APPROACH FOR DISTRIBUTION OF RESOURCES IN STRUCTURAL ANALYSIS SOFTWARE

J.Lindemann*, G. Sandberg and O. Dahlblom

Division of Structural Mechanics, Department of Mechanics and Materials
Lund Institute of Technology, Lund University
P.O.Box 118, SE-221 00 LUND, Sweden
e-mail: strucmech@byggmek.lth.se, web page: http://www.byggmek.lth.se/

Key words: three-tier, DCOM, CORBA, software, structural mechanics, distributed

Abstract. *A sample structural mechanics code is implemented using the Fortran 90 language. This code is encapsulated by DCOM components using the C++ Language. The different components of the code can be transparently placed either locally or remote without changing the client application.*

1   Introduction

A complex hardware product often consists of many exchangeable components. As long as a component fits into the product, the internal implementation can differ. Software components are analogous to hardware components. A Software component consists of a description of its interface and an implementation. Components in programs can be exchanged without the need for recompilation, as long as the program uses the same component interface. The use of components in software development has increased during the last few years. The reason for this is the need to reduce the size of the client programs. When the first client/server systems appeared the client software were often large programs. Most of the processing was done in the client program and the database server was used as data storage. The problem with these systems was the cost of maintaining and installing the client software. The new systems being developed today often use a three-tier approach. A thin client with little or no data processing capabilities is used. Instead of calling the database servers directly, they use a set of components placed on central servers for data processing. These components then access the database servers. The components are often implemented using the DCOM or CORBA standards. The advantage of this approach is that the components can be placed on powerful systems reducing the amount of processing needed at the client. The client software can then be reduced to only handle user interface interaction.

In the present work, a three-tier approach is applied to structural analysis software. The computational parts of analysis codes can be placed on remote servers. Access to the codes can be achieved using components implemented using DCOM or CORBA. The clients can use these components as if they were located on the same machine, making it possible to create integrated programs with transparent access to advanced computational resources.

2   Three-tier applications

Three-tier and n-tier applications emerged from the need to shield the client program from changes at the server side by placing a layer between the client and the server. A detailed history of the client/server architecture is described in [11]. In the following, the three-tier approach as applied to database applications is briefly described. For a more detailed description of how to implement, three-tier applications see for example [9].

The logical three-tier model divides an application in three or more logical components. Each component is responsible for a well-defined task. An application can for example consist of the following components:

- Presentation service responsible for displaying data and editing data.
- Logic/Rules service.
- Database service responsible for storing the data with referential integrity.

The components of the logical model can be grouped together in different configurations to form a physical model.

In the physical two-tier application the logic/rules service are combined with either the presentation service or the database service to form a physical two-tier implementation. When the logical logic/rules service is combined with the presentation service, the client is often called fat client, see Figure 1. When the logic/rules service is combined with the database service this is often called fat server, see Figure 2. The three logical services can also be placed as separate applications at many different servers forming a physical three-tier application. See Figure 3.
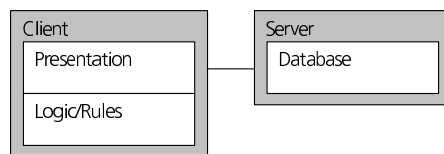
Figure 1 - Fat client
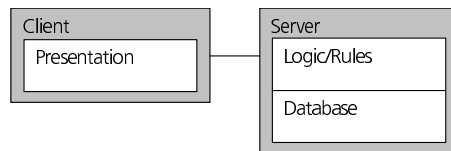
Figure 2 - Fat server
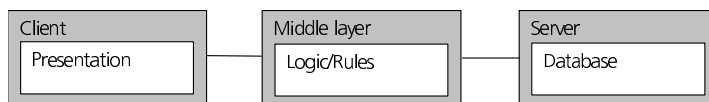
Figure 3 - Physical three-tier application

Today the middle layer often consists of distributed objects implemented using either CORBA or DCOM. These middle layer objects handle the application logic and database connections in an object oriented way. The clients in a physical three-tier application are very light applications, only containing code to display and edit the information it receives from the middle layer. This implementation enables developers to have a greater flexibility in the choice between different hardware and software configurations. It also enables existing systems to be integrated in new program developments. In this scenario, the middle layer also shields the client applications from any changes if the existing system is upgraded in the future.

3 Distributed Computing

Distributed Computing is defined as a type of computing in which different components and objects comprising an application can be located on different computers connected to a network, for an overview see [13]. During the last decades distributed computing has evolved and a number of enabling technologies have been developed. Both DCOM and CORBA implementations make use of these technologies. The following sections describe some of these technologies.

The OSF Distributed Computing Environment [13] is a vendor-neutral set of distributed computing technologies. DCE provides the following services

- Remote Procedure Call
- Directory Services
- Time Service
- Security Service
- Threads Service

These services lay the foundation for the distributed object models DCOM and CORBA.

The remote procedure calls service of DCE [13] enables one program to call a subroutine on a different computer without knowing that the implementation of the subroutine is placed on a server. A first version of RPC was developed in the early eighties by Sun Microsystems as a part of their Open Network Computing architecture (ONC). DCOM as well as many CORBA ORB implementations are based on the RPC service.

4 Distributed Object Computing

There are today two coexisting technologies for distributed object computing DCOM and CORBA.

Microsoft's distributed COM (DCOM) [4] extends the Component Object Model (COM) [12] to support communications among objects on different computers on a local area network (LAN) or the Internet. For a more technical description see [12]. Because DCOM is based on COM many existing COM-based applications can be distributed without modification. DCOM also introduces new facilities for actively controlling remote objects previously not available to COM.

Initially DCOM could only be used on the Microsoft Windows NT and Windows 95 operating systems, but this has changed during the last two years. DCOM is currently implemented on the following platforms:

- Compaq, Tru64 UNIX
- Compaq, OpenVMS Version 7.2

- − Sun Microsystems, Solaris
- − Linux
- − Silicon Graphics, IRIX
- − Hewlett Packard, HP/UX
- − IBM, OS/400 and AIX

CORBA [7] is the Object Management Group's specification for interoperability and interaction between objects and applications. Objects and applications can be placed on any platform and accessed from any platform. CORBA 1.1 was released in 1991 and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0 was released 1994 and specifies how ORBs from different vendors can interoperate. Because CORBA is a specification, it is platform independent. To use the CORBA specification an ORB has to exist for the specified platform. ORB:s exists for almost all existing platforms today.

The implementation described in this paper was performed using the DCOM specification, but it could also have been implemented using the CORBA specification.

5   Distributed Component Object Model (DCOM)

DCOM is widely used for distributed computing on the Windows platform because it is built into Microsoft's operating systems. Most applications using the Component Object Model (COM) can without modification be distributed using DCOM. To use all facilities available in DCOM, modifications to the code are necessary. This chapter will introduce DCOM and the terms used when developing distributed applications with this specification. For a more detailed, description, see [4].

The DCOM specification is language neutral, which makes it possible to write COM and DCOM objects using any language. One of the advantages of this approach is the possibility to encapsulate existing code into DCOM objects.

DCOM/COM objects are accessed through a set of interfaces. The interface is a contract between the client and the object. Once an interface has been released, it can not be changed. If it could be changed, clients using this interface would crash. Functionality is added by adding a new interface to an object. This makes it possible for older clients to access the new object through the old interface.
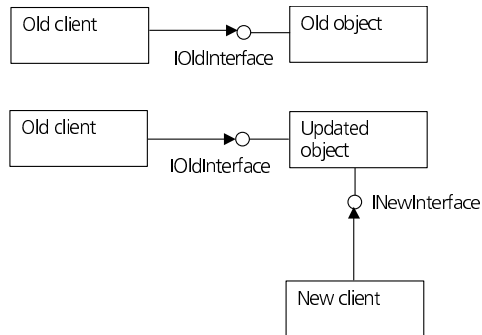
Figure 4 - Extending object functionality

New interfaces can be derived from existing ones using inheritance. Each interface in DCOM is given a globally unique identifier called GUID. The GUID identifier is a 128-bit value generated with a special algorithm guaranteeing it will be statistically unique.

Objects in DCOM implement the functionality of the interfaces. There are three main object types in DCOM. *In-process* object, *Out of process* objects and *Remote* objects.

In-process objects reside in the same process as the client application, see Figure 5. Calls to object methods are done directly through the virtual method table. This means no performance loss when calling an object method. In-process objects are implemented as dynamic link libraries (DLL).
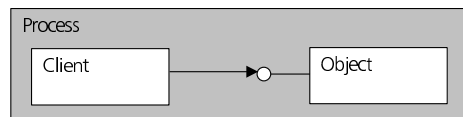
Figure 5 - In-process object

Out of process objects are objects residing in different processes. Clients can not directly call an object in a different process. DCOM handles this by placing a proxy-object in the client process. These proxy-objects then call the DCOM run-time that marshals the parameters over the process boundaries. In the server, a special stub-object calls the actual object itself. Figure 6 illustrates this. The advantage using out of process objects is fault-tolerance. If the object crashes it won't take down the client process. The disadvantage of out of process objects is the time required marshalling the data over the process boundaries. Out of process-objects are often implemented as executable files. In-process objects can also be implemented as out of

process objects by using a special application (dllhost.exe) which loads the object into its process space.
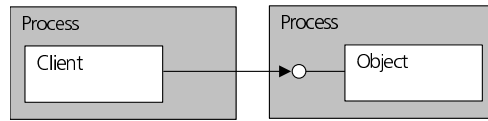
Figure 6 - Out of process objects

Remote objects are objects residing on different machines. These objects are handled in the same way as out of process-objects with a proxy/stub mechanism.
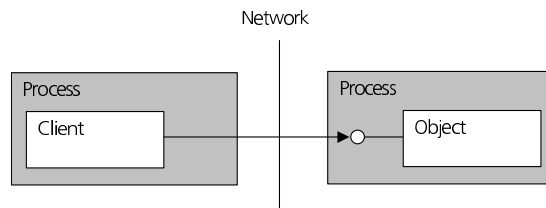
Figure 7 - Remote objects

6   A sample finite element implementation based on DCOM

To illustrate the method for distribution of resources a three-dimensional beam analysis program was chosen as sample implementation. The goal of the sample implementation was to divide it into self contained components. These are then assembled in a visual development system as Borland Delphi 4 [2] or Microsoft Visual Basic 6.0 [6].

The sample application was implemented as a 32-bit Windows application using a three-dimensional user interface implemented in OpenGL [8] see Figure 8. The application was divided into six logical components see Figure 9. By dividing the application into components, the application becomes easier to maintain during development and after. By dividing the middle-layer into more components, the application is more open for different distributed configurations.

Figure 8 - Sample frame application
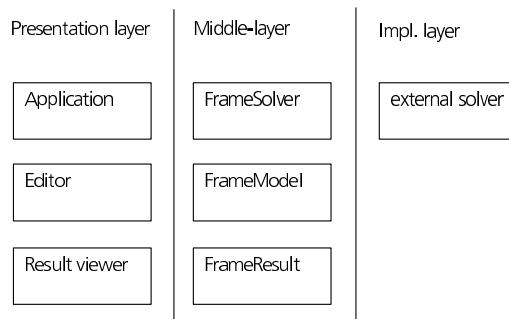
Figure 9 - Logical system components

6.1   Interfaces

In the Frame application, a set of interfaces is used to control the functionality of the different objects. Figure 10 shows the interfaces used in the application. There are two major of advantages of using interfaces in an application:

1. When functionality is added this is done by adding new interfaces to the objects. This makes it possible for older clients to use the old interfaces without modification.

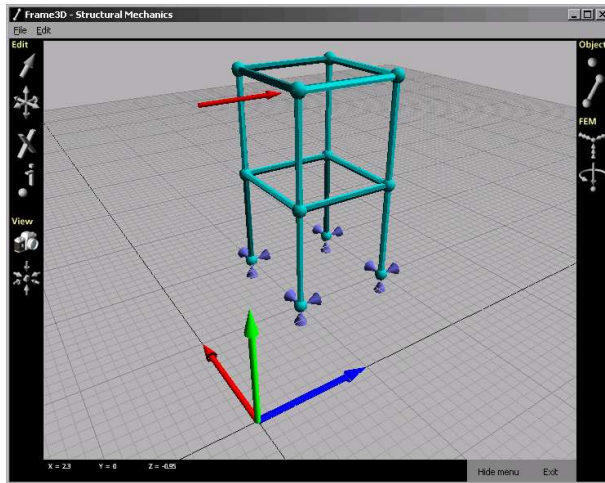2. The middle-layer and implementation layer can be changed completely as long as the new implementation supports the published interfaces clients can use the new implementation without modification.

There are two kinds of interfaces in the Frame application. The first interfaces are the default interfaces returned when an object has been created. From these interfaces, a set of general finite element interfaces can be retrieved to edit finite element data.
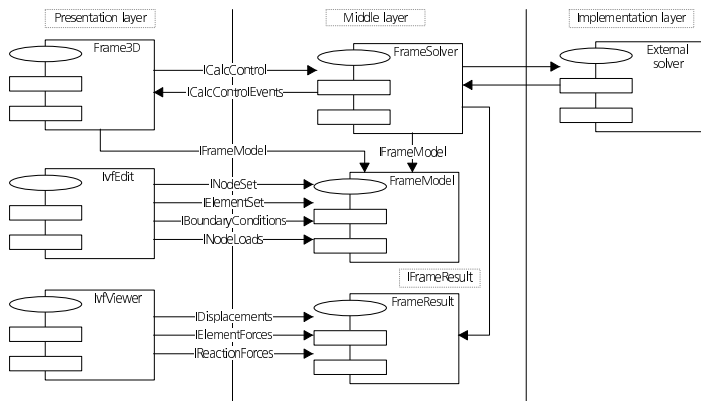
Figure 10 - Interfaces used in the Frame application

Interface	Description
INodeSet	Defines a set of functions for handling a set of nodes.
IElementSet	Defines a set of functions for describing a set of elements.
IBoundaryConditions	Defines functions for setting boundary conditions.
INodeLoads	Defines functions for defining node loads.
IDisplacements	Defines functions for accessing the global displacements
IReactionsForces	Defines function for accessing the global reactions.
IElementForces	Defines functions for accessing element forces.

Table 1 - General finite element interfaces used in the Frame application

If the solver is updated to handle structural dynamic problems, new interfaces can be derived from the above interfaces to implement the new functionality. As an example, *IDynamicNodeLoads* extends the *INodeLoads* interface and *IDynamicDisplacements* extends *IDisplacements*. The new solver can still be used as a static solver by using the old interfaces.

Interfaces can also be officially released with the software, enabling third party software to integrate with the application. This can create new applications that integrate many disciplines. A pre-processor can use the interfaces directly to define the finite element model instead of creating an input file.

6.2   Presentation layer components

The presentation layer objects are responsible for interaction and presentation of the model for the user.

The main application is implemented using Borland Delphi 4 [2] integrated development environment (IDE). Delphi is used to assemble the sample application components into a Windows application. The only code written in the Delphi application is code for managing menus, toolbars and component states. Main functionality is contained in the *IvfEdit* and *IvfViewer* components.

The *IvfEdit*-component handles three-dimensional geometry editing. The component is implemented in C++ as a Microsoft Foundation Classes (MFC) [5] ActiveX component. Drawing is done using a special developed visualisation class library (Interactive Visualisation Framework, IVF) implemented using OpenGL [8]. When the application has been started, the component is given an interface to the *FrameModel* middle-layer component enabling it to update this component automatically. To enhance performance the *IvfEdit*- and *FrameModel* components both maintain an internal representation of the finite element model. This prevents network- or inter-process communication each time a geometric element is added or removed in the *IvfEdit*-component. Updating of the *FrameModel* component is done using the *Load* and *Store* methods of the *IvfEdit*-component. The *Load* method reads the model stored in *FrameModel* and creates an internal representation using IVF. *Store* transfers the internal IVF model to the *FrameModel*-component.
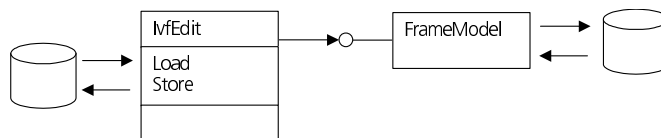

Figure 11 - *IvfEdit* integration with *FrameModel*

The *IvfViewer* -component is responsible for visualising the result database. The component is implemented in C++ as a Microsoft Foundation Classes (MFC) [5] ActiveX component. Drawing is done using a special developed visualisation class library (Interactive Visualisation Framework, IVF) implemented using OpenGL [8]. When the application has been started the component is given an interface to the *FrameResult* middle-layer component enabling it to retrieve information directly from this component.
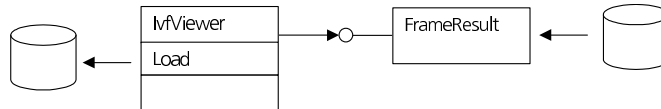
Figure 12 - IvfViewer integration with FrameResult

6.3   Middle-layer components

The middle-layer components shield the client application from the finite element solver details. By using a middle-layer, the finite element implementation can easily be replaced without changing the client applications.

The FrameModel component stores all information needed to describe the finite element model. The component is implemented using C++ and Microsofts Active Template Library (ATL). For a description of how ATL is used, see [10]. ATL is a set of template classes for creating DCOM/COM objects. The template classes have predefined implementations of most of the standard DCOM/COM interfaces that make it easier to create components for DCOM/COM.
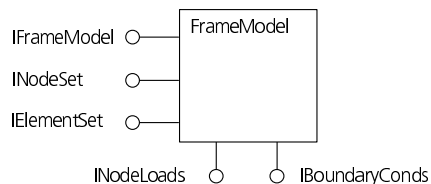
Figure 13 - The FrameModel component

FrameModel maintains an internal class structure describing the finite element model. The classes are exposed with the interfaces *INodeSet, IElementSet, INodeLoads* and *IBoundaryConds.* With these interfaces, it is possible to create nodes, elements, loads and

boundary conditions. In the Frame application, the *FrameModel* component stores itself to disk using a simple textfile. In larger projects, it will probably be necessary to store the finite element model in a relational database to handle large data volumnes.

The *FrameSolver* component manages the finite element solver. The DCOM object is implemented using C++ and Microsoft Active Template Library (ATL). The solver itself can be implemented by calling an external commercial solver or placing custom solver code directly into the component. The last choice is probably the best if a new solver is to be developed. When developing solvers in Fortran, DIGITAL Visual Fortran 6.0 [1] can create special interface modules to enable Fortran code to call DCOM/COM objects directly.
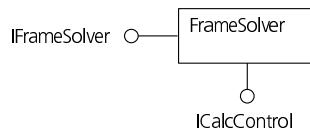
Figure 14 - The FrameSolver component

If the *FrameSolver* component is to be placed on other operating systems than Windows, it is important to develop the component without using any of the Windows user interface routines. When using standard ATL generated objects, this is no problem.

The *FrameResult* component is responsible for managing the results generated from the calculation. This component was implemented using Object Pascal and the Delphi IDE [2]. The results were stored in a relational database. Results from the database are accessed from the interfaces *IDisplacements, IReactionForces* and *IElementForces*.
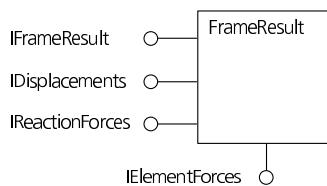
Figure 15 - The FrameResult component

The Microsoft Jet Database engine was used in the Frame application, but if a large amount of data is to be handled a more powerful database engine has to be used. With larger problems terabyte size data are common and places high demands on the database engine.

6.4 Three-tier implementations

Using the components described in the previous section and the three-tier model described in chapter 2 make it possible to configure the application in many different ways. One configuration is to place all components on the local machine. This has the advantage of not relying on any external resources. A disadvantage is that the installation can be quite complex. All components has to be installed and registered. Updating of software often requires a new installation. Figure 16 shows an example of this type of installation.
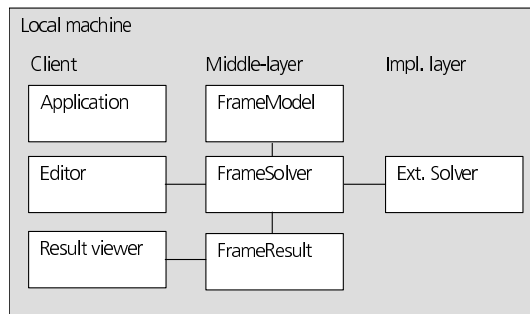
Figure 16 - Local installation

Another configuration is shown in Figure 17. This is a physical two-tier configuration. In this configuration, the middle-layer and implementation layer components are grouped together on a remote machine. The client contains only the application and the visual components. This configuration is typically used if a shared calculation server is installed in the network. The advantage of this configuration is that the client side is easy to maintain and is freed from heavy calculations. The disadvantage of this configuration is that the remote machine is fixed and difficult to modify.

A way of making the configuration more flexible is to divide the components into a physical three-tier solution. In this configuration the middle-layer and implementation layer are placed on different machines. The middle-layer in this configuration can then distribute the work over many machines. The configuration is also more flexible in that the implementation layer can be placed on a super computer and the middle-layer on small Windows NT or Unix servers. Figure 18 shows a physical three-tier configuration.
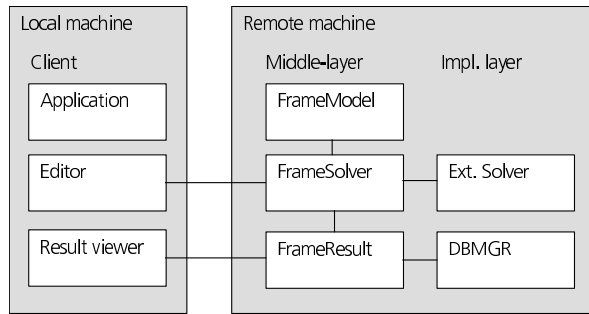
Figure 17 - Physical two-tier configuration
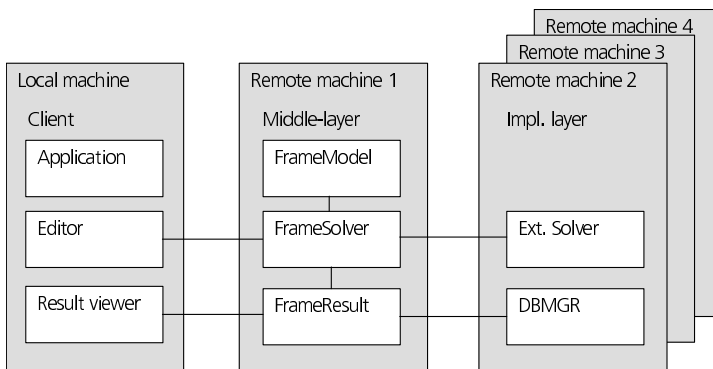
Figure 18 - Physical three-tier implementation

7   Conclusion

Using a three-tier implementation, with interfaces and components creates a very flexible finite element application. The three-tier implementation protects the client applications from changes in configuration and solver design. Components are easily configurable and maintainable which reduced development. By using interfaces when communicating with components, the need to recompile client software when new functionality is introduced in the solver components is reduced. Interfaces can also be published enabling other software to use the finite element application in an effective way.

The DCOM and CORBA specifications also enable new ways to use software. Client software can easily distribute calculations over available workstations. High Performance Computing (HPC) centres would be able to host a set of applications as DCOM or CORBA objects. From a web site users can register themselves as users and download client applications that connect to the objects. This would make high performance computing more available to a wider user group.

References

[1]   COMPAQ DIGITAL Products and Services, DIGITAL Visual Fortran 6.0, http://
        http://www.digital.com/fortran/

[2]   INPRISE Corporation, Borland Delphi 4.0, http://www.borland.com

[3]   Internet.com LLC, PC Webopædia, http://webopedia.internet.com/

[4]   Microsoft Corporation, DCOM Technical Overview, 1996

[5]   Microsoft Corporation, Microsoft Foundation Class Library 4.0: C++ Application
        Framework for Microsoft Windows, 1995

[6]   Microsoft Corporation, Microsoft Visual Basic 6.0, http://www.microsoft.com

[7]   Object Management Group, Inc., http://www.omg.org

[8]   OpenGL The industry's Foundation for High Performance Graphics,
        http://www.opengl.org

[9]   R. Orfali and D. Harkey, Client/server programming with Java and CORBA. – 2nd
        ed., John Wiley & Sons Inc., 1998

[10]   G. Reilly, Developing Active Server Components with ATL, Microsoft Corporation,
        1997

[11]   G. Schussel, Client/Server: Past, Present and Future,
        http://news.dci.com/geos/dbsejava.htm

[12]   S. Willliams and C. Kindel, Microsoft Corporation, The Component Object Model: A
        Technical Overview, 1994

[13]   The Open Group, http://www.opengroup.org/dce

Paper II

Using CORBA Middleware in Finite Element Software

Using CORBA middleware in finite element software

J. Lindemann, O. Dahlblom and G. Sandberg

Division of Structural Mechanics, Lund University
strucmech@byggmek.lth.se

Abstract. Distributed middleware technologies, such as CORBA can enable finite element software to be used in a more flexible way. Adding functionality is possible without the need for recompiling client code. Applications and libraries can expose their functionality to other applications in a language neutral way, enabling a more direct and easy transfer of data, without the need for intermediate input and output files. The CORBA software components can be easily configured and distributed tranparently over the network. A sample structural mechanics code, implemented in C++ is used to illustrate these concepts. Some future directions, such as placing CORBA enabled finite element software on HPC centres are also discussed.

1 Introduction

A complex hardware product often consists of many exchangeable components. As long as a component fits into the product, the internal implementation can differ. Software components are analogous to hardware components. Components in programs can be exchanged without the need for recompilation, as long as the component interface is unchanged. The use of components in software development has increased during the last few years. The reason for this is the need to reduce the size of the client programs. When the first client/server systems appeared, the client software were often large programs. Most of the processing was done in the client program and the database server was used as data storage. The problem with these systems was the cost of installing and maintaining the client software. New systems developed today often use a thin client with little or no data processing capabilities. Instead of calling the database servers directly, they use a set of components placed on central servers for data processing. These components then access the database servers. The advantage of this approach is that the components can be placed on powerful systems, reducing the amount of processing needed at the client. This approach has been successfully applied to database applications. It is of interest to apply this technique to analysis software as well. Using the technique of distributed computing, clients can use components as if they were located on the same machine, making it possible to create integrated programs with transparent access to computational resources, such as available workstations on the network or resources at High Performance

Computing (HPC) centres. This would make high performance computing more available to a wider user group.

The present work describes structural analysis software, where the computational parts of analysis codes can be placed as components on remote servers. Before describing the structural analysis code, a brief overview of client/server architecture will be given.

2 Client/server architecture

Three-tier and n-tier applications emerged from the need to shield the client program from changes at the server side by placing a layer between the client and the server. The history of the client/server architecture is described by Schussel [25]. For a more detailed description over the client/server architecture, see Orfali and Harkey [18]. The logical three-tier or n-tier model divides an application into three or more logical components. Each component is responsible for a well-defined task. In a database application there would be a presentation layer for displaying data and modifying data, a logic and rules layer and a database layer responsible for storing the data.

The components of the logical model can be grouped together in different configurations to form a physical model. One of the most interesting combinations of the logical model is when the three logical services are placed as separate applications on different computers, forming a physical three-tier application. This implementation enables developers to have a greater flexibility in the choice between different hardware and software configurations.

3 Distributed computing

Distributed computing is defined as a type of computing in which different components and objects comprising an application can be located on different computers connected to a network; for an overview see [16].

Currently, there are three coexisting technologies for distributed object computing DCOM [2], Java Remote Method Invocation RMI [24] and CORBA [1]. Microsoft's distributed COM (DCOM) extends the Component Object Model to be used over the network. RMI or Remote Method Invocation [24] is a distributed technology based on the Java language. CORBA is the Object Management Group's [1] specification for interoperability and interaction between objects and applications. Objects and applications can be placed on any platform and accessed from any platform.

This paper describes an implementation in CORBA. In a previous paper [12] a DCOM based implementation has been studied.

DCOM is mainly used on Microsoft compatible platforms, but using third-party products, it can be ported to most Unix platforms. This technology enables Java objects to communicate transparently over the network. To use the CORBA specification there has to be an ORB (Object Request Broker) for the specified platform. There are ORBs for almost all existing platforms today.

4   CORBA

4.1   Concepts and Terminology

To describe a CORBA based implementation, it is important to understand some
terminology and concepts of a CORBA implementation. Some of the more im-
portant concepts and terminology is shown below. A more thorough description
can be found in Henning and Vinoski [10].

- A *client* is an entity that invokes a request on a CORBA object.
- A *CORBA object* is a "virtual" entity capable of being located by an ORB
  and having client requests invoked on it.
- A *server* is an application with one or more CORBA objects.
- An *object reference* is a handle used to identify, locate and address a CORBA
  object. Object references is the only way for a client to access CORBA
  objects.
- A *servant* is a programming language entity that implements one or more
  CORBA objects.

Communication in CORBA is done by a client invoking requests on a CORBA
object through either a statically linked stub in the client application or through
the dynamic invocation interface (DII). The requests are dispatched to the local
ORB which in turn dispatches these requests to an ORB on the remote ma-
chine. The remote ORB then dispatches the request to an object adapter, which
then directs the request to the servant implementation code. Figure 1 shows an
overview of the CORBA architecture.

4.2   Interface definition language

To access a CORBA object the client must know which methods and properties
it contains. This description is called an interface. To describe such interfaces
CORBA uses the Interface Definition Language (IDL). In this language the
object interfaces are described. Using a separate language for describing the
objects makes CORBA language neutral. This enables CORBA applications to
be implemented in a variety of different languages. To implement CORBA clients
and objects the IDL definition is compiled using an IDL compiler. This compiler
takes the interface definition and generates the implementation code for both
client and server, in the desired implementation language.

The following code shows an example of a simple IDL interface, declaring an
interface to an `Echo` object. In this case the object echoes the string word back
to the calling client.

```
interface Echo {
    string Shout(in string word);
}
```
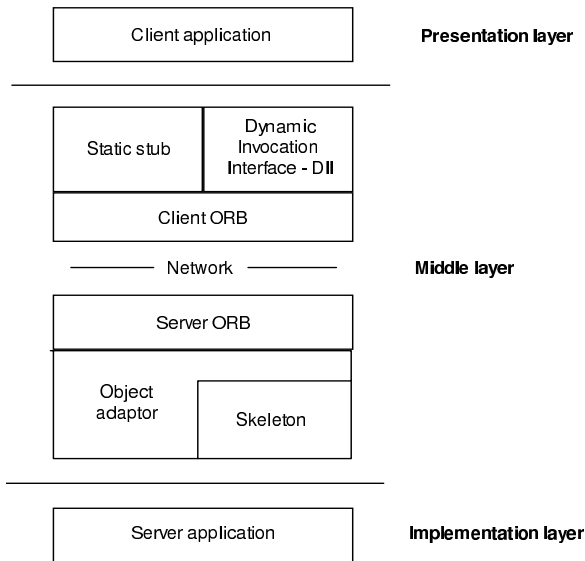
Fig. 1. CORBA architecture

Compiling this example using a C++ IDL compiler, will generate a header file and an implementation source file for accessing the object described from a C++ based application and the skeleton code for implementing the servant object in C++.

In Orbacus [17] the IDL compiler implements the interface in the `Echo` object, using C++ classes. The generated client code is shown below.

```
class Echo : virtual public CORBA::Object {
    Echo(const Echo&);
    ...
public:
    ...
    char* Shout(const char* word);
};
```

The different files generated by the C++ IDL compiler are shown in figure 2.

4.3   Name service

One of the biggest benefits of CORBA is location transparency. Information about server location is often not included in the client application. This makes it easy to configure a client/server setup. A client only needs an object reference
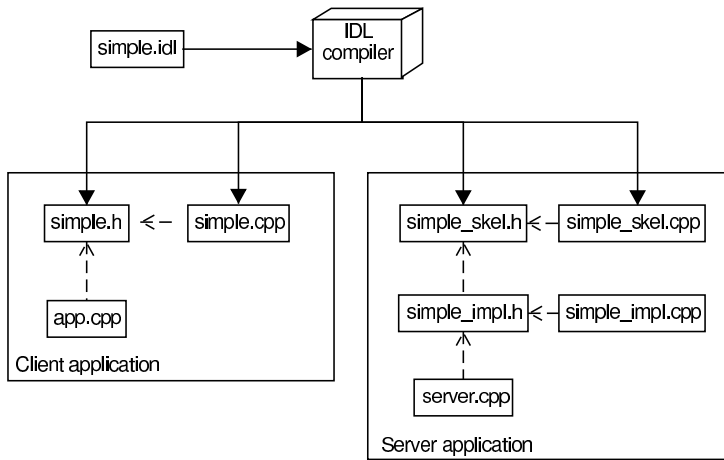
Fig. 2. Relationships between the files generated by the C++ IDL compiler

to connect to an object. Object references are unique identifiers, which also include information about the location of objects. To connect to objects the client needs a way of retrieving an object reference. Before the introduction of CORBA 2.3, object references were often transferred using files over a network file system or using a non-standard method of name lookup. In CORBA 2.3 a name service was introduced. The name server stores object references in a human readable form. When a server is started, it creates an entry in the name server for the object reference. The client then queries the server by name to receive the object reference. See Figure 3. By using a name server, client/server configuration can be done transparently. Name server location is the only thing that has to be configured for the servers and the clients. Clients and servers get the location of the name server by specifying special command line options.
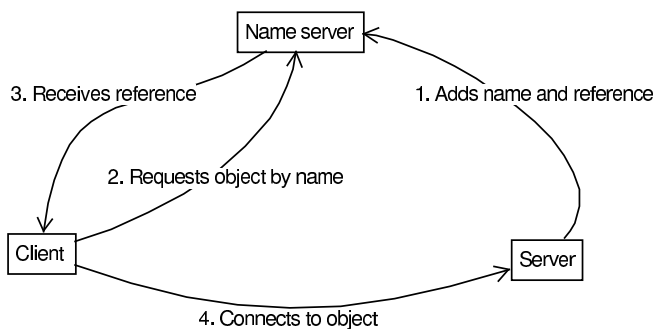
Fig. 3. Name server lookup

49

4.4 Object creation and destruction

Before request to an object can be made, the object implementation (servant) must be instantiated and activated. In CORBA this is done by the object adapter. Earlier CORBA specifications only included a limited basic object adapter (BOA). To enhance the functionality of this object adapter many ORB vendors added non-standard extensions. The consequence of this was that the server side of a CORBA application became ORB dependent. With CORBA 2.3 this limitation was removed by the introduction of the Portable Object Adapter (POA).

Different types of policies for the creation and destruction of objects can be specified using lifetime policies for the portable object adapter (POA) in CORBA. Figure 4 illustrates the typical lifetime of a CORBA object. The default policy is `TRANSIENT`. In this policy the object can not be reactivated, when it has been deactivated. The object reference of a `TRANSIENT` object is only valid when the object is active. The `PERSISTENT` lifetime policy enables objects to be activated and deactivated multiple times. This requires that the object servants are able to store their state in a persistent form between the activations.
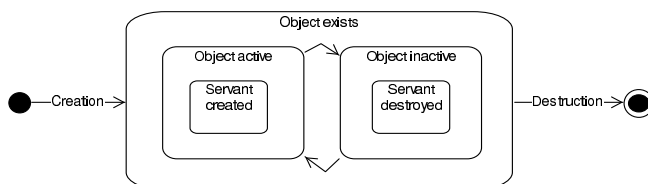
Fig. 4. Object creation and destruction

Because CORBA is a distributed technology, the creation of objects must be handled in a different way than it is handled when creating local objects. In a CORBA system, objects are created by special factory objects.

The destruction of a CORBA object is not done by the factory, instead a special method is declared in the object interface for removing the object. If the factory was responsible for destroying the object, the client referencing the object would also have to reference the factory when destroying the object. This would be quite complex if the object reference has been passed from object to object. The process of creating and destroying is discussed in detail in Henning and Vinoski [10].

## 5 CORBA in finite element software

Most finite element applications communicate using files. The input model is described in a text file in some form. Generated results are often stored in a

binary output file or in a database. If other applications, such as pre and post processors are integrated with the finite element application they have to generate and read these files. This generates a lot of extra steps to integrate existing finite element codes. Using CORBA the inner object model and functions of the application can be exposed directly to other applications, which can access them either locally or remotely. To communicate with the CORBA enabled application, a wrapper for the given implementation language is generated from an interface definition as given in the IDL specification. The exposed objects and functions are accessed from the client applications as local objects and functions in the clients native implementation language. Element coordinate lists and topology can be sent directly to the finite element application, over the network or in the same memory space with good performance. This approach can also be implemented using DCOM [2] which has been done by Lindemann et al. [12], Larsson [11] and Dolenc and Duhovnik [3].

The normal way of distributing finite element applications, is to install the software on a remote server and letting the users log in remotely and execute the application. If the server is behind a firewall, a distributed file system can be used to access the generated files. The process can be simplified using scripts and remote execution, but the process is still quite complicated and the applications still communicate using files. CORBA enabled applications can access resources transparently over the network or locally. A CORBA enabled pre processor does not need to know the location of the finite element application when compiled. When the application is executed, it queries a CORBA name server which then provides the location of the finite element application to be used.

Functionality of the CORBA application is defined using the interface definition language IDL. To interface a pre-processor with a CORBA enabled finite element application, the IDL file is compiled, generating the necessary communication code and interface functions and classes. The client application is then recompiled and linked with the IDL generated code. To make this work, existing finite element applications must be CORBA enabled. Most finite element applications today are not CORBA enabled, making it difficult to integrate them into other CORBA based systems. To make interaction of CORBA enabled finite element applications a reality, a set of standardised IDL definitions must be agreed upon. A standardised set of interfaces enables component oriented applications, where the different application components can be exchanged in an easy way. To integrate finite element codes today in CORBA based systems is to use CORBA wrappers. This approach is used by Forkert et al. [27] in the TENT framework which is a integrated simulation environment. This framework uses CORBA wrappers and a set of translators to create input files and process results. Another implementation of CORBA is done by Frisch and Ertl [5] where a finite element solver is integrated with a post processing tool. Figure 5 shows an example of a component based finite element system.

The trend today is that many applications are becoming web-based. Many companies are using an internal web for distributing knowledge in the organisation. A CORBA based finite element application can effectively be used together
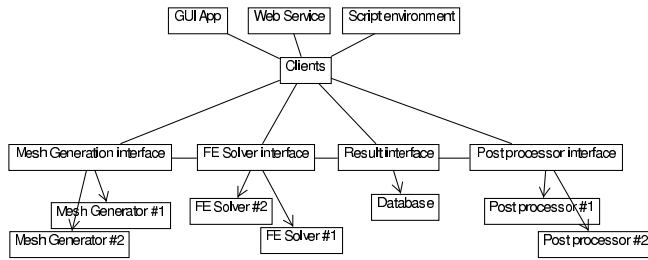
Fig. 5. Interfaces with different FE applications

with a web-based application. Java is CORBA enabled by default, enabling a Java based web application having access to CORBA based services. An example of this could be a web-based engineering tool providing support for engineers making design decisions. The tool first uses a parameterised model to give quick answers within certain parameter ranges. If the parameters are outside these ranges a simulation is initiated using a remote CORBA enabled finite element application. The results are then stored in a database to be reused later on. The advantage of using CORBA in this application is flexibility. The client does not communicate directly with the finite element application but through the CORBA server. The CORBA server can choose to execute the finite element application on the same machine or it can delegate the execution to another machine. This can be done without changing anything in the client code. Figures 6 and 7 show an example of how an application of this kind can be constructed.
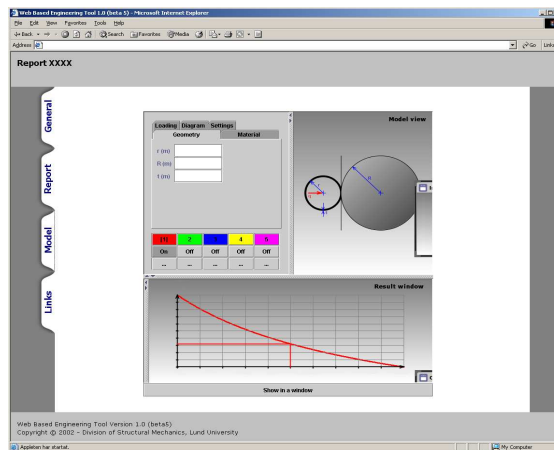
Fig. 6. Virtual Engineering Tool
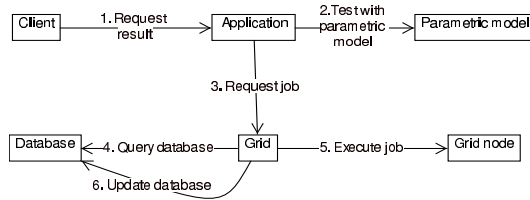
Fig. 7. Engineering tool execution process

5.1 CORBA as a scripting interface to FEA applications and libraries

A powerful concept that can be used together with CORBA enabled application is scripting. There exist a number of very powerful scripting languages such as Python [22], Ruby [23], Perl [19] and Tcl/Tk [21], that can be used together with CORBA. Using CORBA with a scripting language, a parametric study of a problem can easily be implemented without the need to create input file and read result files. CORBA applications are accessed as standard Python objects, all input generation and result processing can be done directly in the scripting language, communicating directly with the finite element application. This works with local objects as well as remote objects. Interfacing scripting languages with CORBA objects can be problematic, because it often requires using an ORB from another vendor or developer. There exist differences between different ORBs today which can make it difficult connecting these with each other. These differences will probably become less important as the CORBA standard evolves.

The following code excerpt illustrates how a parametric study can be done using CORBA and Python. The CORBA ORB used with Python is Fnorb [7].

```
################################
# Python CORBA client example.  #
################################

import sys

# Fnorb modules.

from Fnorb.orb import CORBA

# Stubs generated by 'fnidl'.

import FEApplication

def main(argv):
```

```
    print 'Initialising the ORB...'

    # Initialise the ORB.

    orb = CORBA.ORB_init(argv, CORBA.ORB_ID)
    server = ... get object reference somehow ...

    # Do a parameter study

    parameter = 0.2

    while parameter < 3.2:
        server.setWidth(parameter)
        server.set{... additional parameters ...}
        server.execute()

        # process results

        server.getMaxStress(...)

        .
        .
        .

        parameter = parameter + 0.2

    return 0

if __name__ == '__main__':
    sys.exit(main(sys.argv))
```

5.2 CORBA as a language neutral description for libraries

CORBA objects are designed to be accessed remotely, but if the objects are located in the same process space any calls to object methods are treated as normal function calls with approximately the same performance as a normal function call. The GNOME desktop environment [8] uses CORBA to provide a language interface to the different libraries. Interfaces are described using IDL and from these, interface code for languages such as Python [22], Ruby [23], Perl [19] and Tcl/Tk [21] and C/C++ can be generated. This is similar to the approach Microsoft is using in the Component Object Model(COM) [2]. COM/DCOM uses a modified version of the IDL language to describe the interfaces for the objects and components. This enables the use and interoperability of the COM/DCOM objects from all supported languages. The approach described above can also be effectively used to encapsulate PDE and finite element libraries into CORBA libraries with interfaces described in IDL. The advantage of this is that the libraries become language neutral. Users of the libraries can choose whatever language is supported by the their CORBA ORB and generate the interface au-

tomatically from the IDL interface files. The method is similar with the method used by the SWIG tool [20]. SWIG or Simplified Wrapper and Interface generator uses a special interface file `.i`-file to define an interface to a C or C++ library, which is then used to generate interface code for different higher level scripting languages. The advantage of using CORBA is that the library definitions can be used remotely as well.

5.3 Performance

An important factor to consider when implementing a CORBA based finite element application data is transfer performance. Often large data sets of several gigabytes have to be transferred when doing finite element simulations. There are some factors to consider when designing a CORBA interface. The cost of each request on a CORBA object is determined by the latency and marshalling rate. The latency is the cost of sending a message. The marshalling rate is the cost of sending the input and return variables. For a more detailed discussion see chapter 22.3 in Henning and Vinoski [10]. One of the most critical factors for performance is the latency. The latency time of invoking a request on a CORBA object is approximately 500-5000 times higher than doing a function call in C++. For more detailed study of CORBA performance and scalability see Gokhale and Schmidt [26] and the OMG whitepaper [14]. Finite element input data should be transferred in few CORBA requests, so that the latency overhead is minimized. The mashalling rate is also an important factor determining the performance of the finite element application. To determine the optimal block size and transfer speed when transferring data between a client and a server, a test application was written. In the application blocks of sizes 2^0 to 2^{27} are sent between a client and a servant on a 100 Mbit/s network. When transferring a block using CORBA the servant application also has to allocate memory for the return variable. The size of the array to be received is not known by the servant which has to allocate the array in some way. The way this is done by the CORBA ORB is not defined. In ORBacus this is done using a block allocation scheme for the array. To measure the real transfer speed the application tests the time it takes to allocate the array it sends. The test application is written in C++ using the ORBacus [17] CORBA ORB. A special test function was defined in IDL containing just a single CORBA `sequence<octet>`. The simulation of allocation is done by sequential adding of values to the block. This can be thought of as simulating the bytes arriving and being added to the sequence on the server side. To get an accurate allocation time the procedure is repeated 100 times. Mashalling rate is measured by calling the test function with the allocated block. This is done 20 times to get a good result. A test application using normal socket communication was implemented to compare the CORBA application to an alternative network implementation. Figure 8 shows the mashalling rate of different block sizes. The solid line with the + symbols indicates CORBA mashalling rate without consideration of allocation time. The dashed line shows the rate with allocation time considered. The dotted line marked with squares shows the standard socket applications throughput.

The dotted line marked with the symbol (*) is the peak transfer rate over a 100 Mbit/s network.
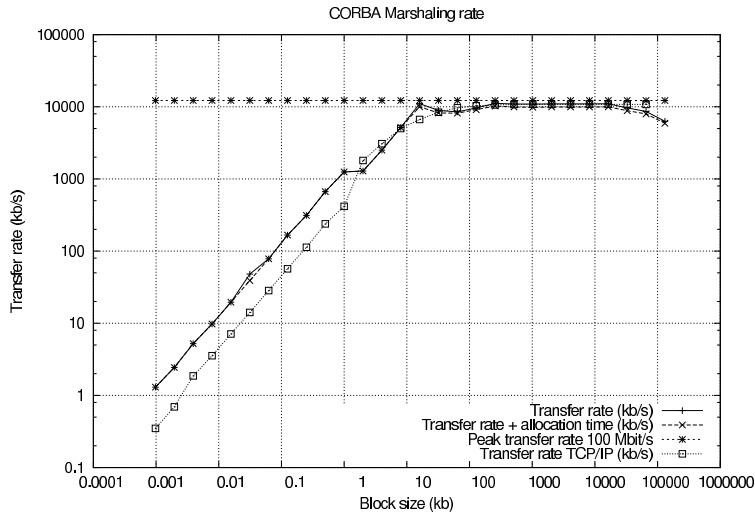
Fig. 8. CORBA Mashalling rate of different block sizes

From the diagram it can clearly be seen that blocks larger than 10 kb can be effectively sent using CORBA with high mashalling rates. A block of 16384 kb achieves a transfer rate of 10998 kb/s. The TCP/IP socket application achieves a transfer rate to 10817 kb/s. The value achieved for the TCP/IP application would probably be slightly higher if a more efficient implementation was used. The decrease in transfer speed shown at the end of the curve at 32768 kb is probably due to memory swapping at the server machine, which was a PIII 967 MHz machine with 256 Mb memory. The allocation test was done on an AMD 1700+ machine with 512 Mb of memory. The source code for the test routine is shown below.

```
void doBandwidthBenchmark(Node::FileTransfer_var &fileTransfer) {
    int i, j, k;
    Node::TFileBlock block;

    for (i=0; i<28; i++)
    {
        int blockSize = pow(2,i);
        double duration;
        double durationTotal;
        double allocTime;
        clock_t start, finish;
```

56

```
//
// Measure allocation speed
//

Node::TFileBlock* blockTest;

start = clock();
for (k=0; k<100; k++)
{
    blockTest = new Node::TFileBlock();
    for (j=0; j<blockSize; j++)
    {
        blockTest->length(j+1);
        (*blockTest)[j] = j;
    }
    delete blockTest;
}
finish = clock();
allocTime = (double)(finish - start) / CLOCKS_PER_SEC / 100.0;

//
// Allocate block to transfer
//

block.length(blockSize);
block[blockSize-1] = 42;

//
// Measure transfer time
//

start = clock();
for (j=0; j<20; j++)
    fileTransfer->blockTest(block); // blockTest(...)
                                    // is the test function
finish = clock();

//
// To obtain the transfer rate the allocation time has to
// be subtracted from the time measured, because the
// servant will allocate a TFileBlock on the server side
//

duration = (double)(finish - start) /
    CLOCKS_PER_SEC / 20.0 - allocTime;

durationTotal = duration + allocTime;
printf( "%f %f %f %f %f %f\n",
    block.length()/1024.0,
```

```
                block.length()/duration/1024.0,
                block.length()/durationTotal/1024.0,
                duration,
                durationTotal,
                allocTime
        );
    }
}
```

5.4 CORBA Interface design for distributed applications

The object-oriented nature of CORBA makes it possible to create an object
model which is very expressive. All features such as polymorphism, data cap-
suling, inheritance can all be applied to a CORBA object. A remote object is
accessed in the same way as its local counterpart, enabling the creation of very
advanced and complex interfaces. Using the same guidelines for object-oriented
design as in other object-oriented languages such as C++ or Java can be prob-
lematic. As discussed in section 5.3 a request on a CORBA object is 500-5000
times slower than making a function call in C++. This has to be considered
when designing CORBA interfaces.
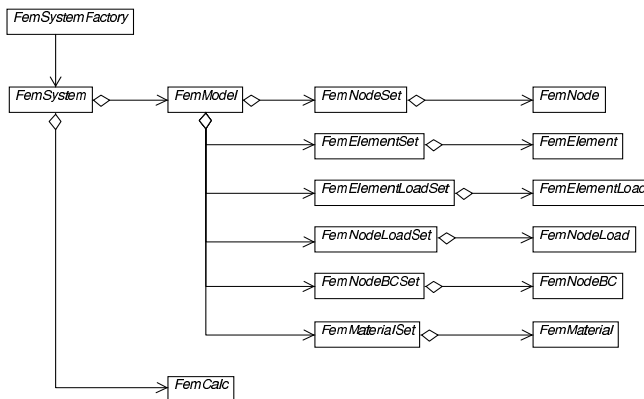
Fig. 9. Interfaces in the ObjectiveFrame application

Most finite element applications today deal with a large number of nodes
and elements, producing large amounts of data. CORBA interfaces for finite
element applications must take this into concideration. In a previous application
ObjectiveFrame, a very expressive CORBA interface was developed, as shown
in figure 9. The interface made it easy to transfer and communicate with the
CORBA objects, but it was not very efficient. To define a node set with the
CORBA object several requests had to be made, as shown in the code below.

```
DFEMC::FemNode_var dfemcNode = dfemcNodeSet->getNode();

dfemcNodeSet->first();

for (i=0; i<nodeSet->getSize(); i++)
{
    // node and nodeSet are a C++ objects
    CFemNode* node = nodeSet->getNode(i);
    node->getCoord(x, y, z);

    // dfemcNode and dfemcNodeSet are CORBA objects
    dfemcNode->setCoord(x, y, z);
    dfemcNodeSet->next();
}
```

To set the coordinates of a node set, a special node interface is retrieved (`dfemc-NodeSet->getNode()`). This interface is used to set the properties of the current node. Changing to a different node is done by calling the `next()` and `previous()` methods of the node set interface. This interface design requires two CORBA requests per node, which is not very efficient. To modify the design to support efficient data transfers, additional methods has to be added to the node set interface to support block transfer of nodes. The code would then change to:

```
// nodeSet is a C++ object
TNodeCoordArray array = nodeSet->getNodeCoordsArray();

// dfemcNodeSet is a CORBA object
dfemcNodeSet->setNodeCoordsArray(array);
dfemcNodeSet->set...
```

This design still has the possibility to access individual nodes on the CORBA node set using the node interface.

CORBA has advantages even if the interfaces are very simple and shallow. Creating distributed network applications using conventional TCP/IP socket programming can be difficult and error prune. A lot of testing is required to create a stable network protocol. Transferring data between different hardware platforms will also require the programmer to take care of the different byte-orderings existing on these. Multiuser systems is also an issue making the applications even more complex, requiring threaded code. Using CORBA, networking code is already implemented in the ORB. Byte ordering is also automatically handled and the transferred data can be assumed to be correct on every hardware platform. Most CORBA ORBs also handle the threading issues automatically.

5.5   CORBA in GRID computing

An important area where CORBA can be used is GRID computing. Grid Computing is a concept of creating grids of computational and storage resources, in the same way as the the world wide web is a grid of information resources.

CORBA can effectively be used as the glue connecting the different resources located at geographically different locations. A computing cluster can have a CORBA based interface for monitoring, job submission and control. By providing each cluster with a CORBA interface it is relatively simple to connect these together creating "Meta"-clusters. Figure 10 shows an example of a CORBA enabled grid system.
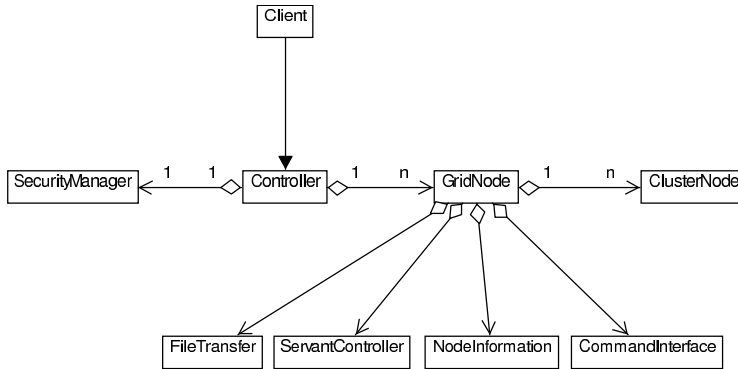
Fig. 10. Example of a CORBA based grid

6 Example of a Finite element CORBA implementation

The educational software ForcePAD [6] was modified to use a CORBA based finite element solver. The ForcePAD application is an intuitive tool for visualising the behaviour of structures subjected to loading and boundary conditions. ForcePAD uses a bitmap canvas on which the user can draw the finite element model using standard drawing tools. When the calculation is executed the bitmap image is transferred to a finite element grid, which is then solved. The main window is shown in Figure 11. The application consists of four components divided into three layers, as shown in figure 12. The user interface is responsible for interactively defining the problem. The ForcePadSolver component contains the interfaces used to describe the finite element model used in the application. The name server components handle the location of available CORBA ForcePAD-Solver components in the network. The FE solver components are responsible for executing the calculations. By providing the functionality of the application in a component based form, the application can be configured and maintained in a more flexible way.
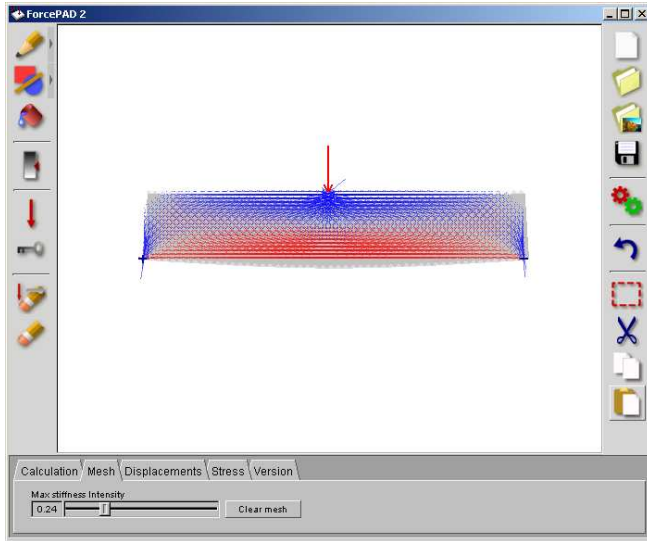
Fig. 11. Sample CORBA application
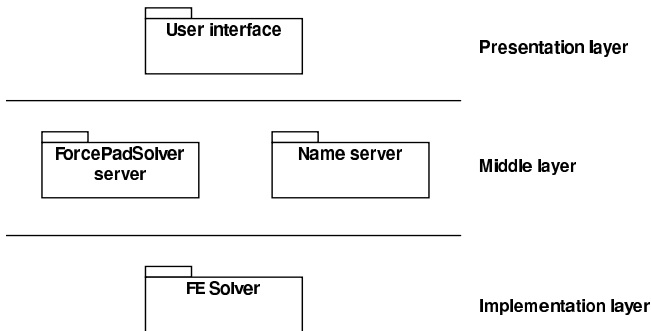
Fig. 12. Application components

6.1 ForcePadSolver server

The middle layer of the application is implemented in a single server. The ORB used in the implementation is ORBacus [17], which is a commercial ORB available with source for multiple platforms, including Microsoft Windows and many Unix dialects. For non-commercial use It can be used without cost. The FE solver is implemented in C++ using the newmat09 [13] library, which is freely available with source code. In this version of the application, the FE solver is statically linked into the ForcePadSolver server, but it is possible to implement the FE Solver as a separate CORBA object or use a standard FE code.

To make this example easier to follow, the interface of the ForcePAD server is made deliberately shallow. A more object-oriented interface as described in section 5.4 is probably to be preferred. To enhance the network performance the interface transfers the entire finite element grid in one large block. The full IDL source code is found in appendix A. Figure 13 shows the interfaces used in the ForcePadSolver server. The main interface in the server is the `FemSystem`
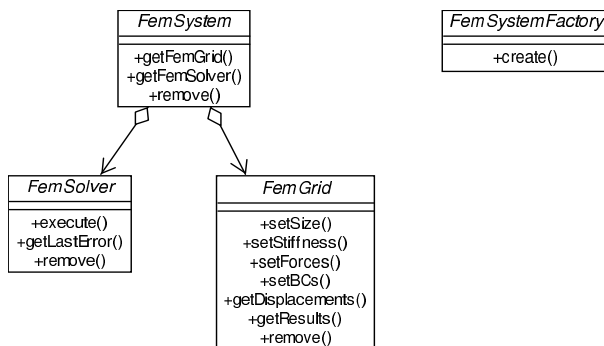
Fig. 13. Interfaces used in the finite element server

interface. Every time a client connects to the server it will create this object, using the `FemSystemFactory` factory object. The factory object is instantiated and registered in the name server when the server is started. The `FemSystem` object, when instantiated will create an instance of a `FemSolver` object and a `FemGrid` object. These objects are returned from the `FemSystem` object. A ForcePadSolver server can hold one instance of `FemSystem` objects for each client connected to the server, as shown in Figure 14.

The code below shows how a `FemSystem` object is created from C++ using the `FemSystemFactory` object.

```
femSystemFactory = ... Get from name server ...
femSystem = femSystemFactory->create();
femGrid = femSystem->getFemGrid();
femSolver = femSystem->getFemSolver();
```

The `FemGrid` object defines the finite element model and the `FemCalc` is used to control the calculation of the finite element model.

To reduce the marshalling times for the FE model, data will mainly be transferred using the CORBA data type `sequence`. This data type is a dynamic array of a specified type. The following code illustrates a typical data transfer from the client to a CORBA object in the ForcePAD client application. The complete client code can be found in appendix C.
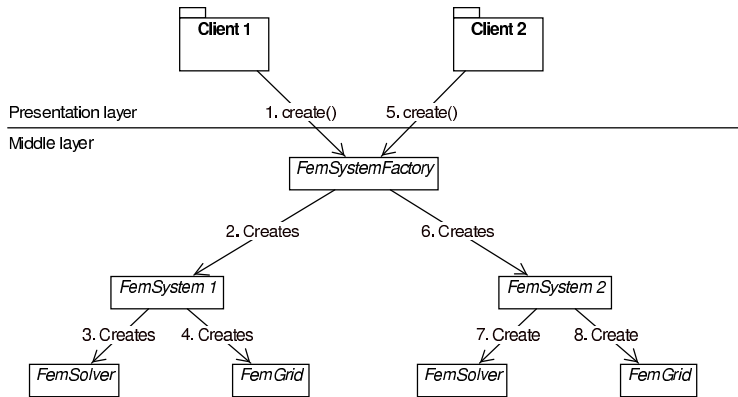
Fig. 14. Object creation using the FemSystemFactory object

```
// CORBA defined datatype:
//      typedef sequence<double> TStiffnessVector;

ForcePadSolver::TStiffnessVector stiffnessVector(nStiffness);
stiffnessVector.length(nStiffness);

// Transfer internal fem model to stiffnessVector

l = 0; float value;
for (i=0; i<rows; i++)
    for (j=0; j<cols; j++)
        for (k=0; k<2; k++)
        {
            value = m_femGrid->getGridValue(i, j, k);
            stiffnessVector[l++] = (double)value;
        }

// Invoke request on femGrid CORBA object

femGrid->setStiffness(stiffnessVector);
```

When all input data has been transferred to the CORBA object `FemGrid`, the finite element model can be solved. The execution of the finite element solver is controlled by the `FemCalc` object. The following code from the client application shows how the calculation is initiated:

```
femSolver->execute();
error = femSolver->getLastError();
```

In the ForcePadSolver server the `execute()` method is implemented as a blocking call. This means that the execution of the client application will wait

until the server is finished. To solve this, the `execute()` could be implemented as an asynchronous method call in CORBA. Additional methods for monitoring the execution would have to be added to the interface as well.

The results from the calculation are also retrieved using the CORBA data type `sequence`. The difference is that the `sequence` vectors now are preallocated and must be transferred back to the C++ class `CFemGrid`. The following client code shows how the results are retrieved from the `FemGrid` object.

```
// CORBA defined datatype:
//          typedef sequence<double> TDisplVector;

ForcePadSolver::TDisplVector* displacements;

// Invoke request on femGrid CORBA object

femGrid->getDisplacements(displacements);

// Store displacement values in local class m_femGrid

m_femGrid->setDisplacementSize(displacements->length()); for
(i=0;i<displacements->length(); i++)
    m_femGrid->setDisplacement(i+1, (*displacements)[i]);

// We are responsible for deleting the return values

delete displacements;
```

The lifetime policy used in the ForcePadSolver server is `TRANSIENT`. A calculation in ForcePAD does normally not execute more than a few seconds, so the policy `PERSISTENT` will not be necessary in this case, it is better suited for applications executing over several days. The client applications can then connect and disconnect to object during the execution.

6.2 Server implementation

The ForcePAD solver server is implemented as a C++ console application using the ORBacus [17] ORB. A skeleton implementation for the server is generated using a special switch in the ORBacus IDL compiler.

The compiler generates a special skeleton implementation class from which the implementation classes are derived. A sample implementation class is also generated. The generated skeleton classes handle the requests from the clients and dispatch them to the implementation class. The skeleton class itself is derived from a number of classes in the ORBacus ORB.

To handle object creation and destruction automatically, each servant is also derived from the `RefCountServantBase` base class. This class implements a reference counting scheme which automatically destroys the object servant when there are no connections to the object. Depending on the implementation, more complex schemes of object creation and destruction can be implemented, see [10]

for more details. Part of the source code for the server implementation can be found in appendix B.

The process of executing a calculation starts with a request to the `FemSolver` method `execute()`. The `FemSolver` reads the input model from the `FemGrid` object and assembles the finite element model. The solver from the `newmat09` [13] is then called. When the solution is found the results are stored back in the `FemGrid` object. The results are now available to the client application.

6.3   Client/server configurations

The easiest configuration of the finite element system is to install the client application together with the ForcePADSolver server and the finite element solver on a single computer, see Figure 15. This configuration is typically used to do calculations that fit into the memory of the local machine.
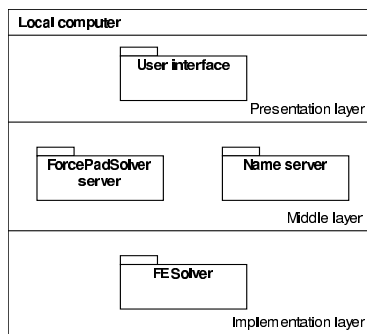
Fig. 15. Local configuration

In the first distributed configuration, the middlelayer and implementation are moved to a separate computer. This configuration requires the server to be able to run a CORBA ORB. If the server running the finite element solver does not support running an ORB, the middlelayer can be placed on a separate computer. Execution of the finite element solver can then be done using `rexec`, `rsh` or `ssh` utilities. Figure 16 shows two of the possible configurations. Many more configurations are possible. By providing location transparency, the CORBA objects can be configured in almost any way without needing to recompile the clients and the servers.

6.4   Client application

To create a platform independent application, ForcePAD uses the fast light toolkit (FLTK) [4]. FLTK is a lightweight user interface toolkit written in C++. The toolkit can be used on Windows 9x/NT/2000/XP and most Unix dialects
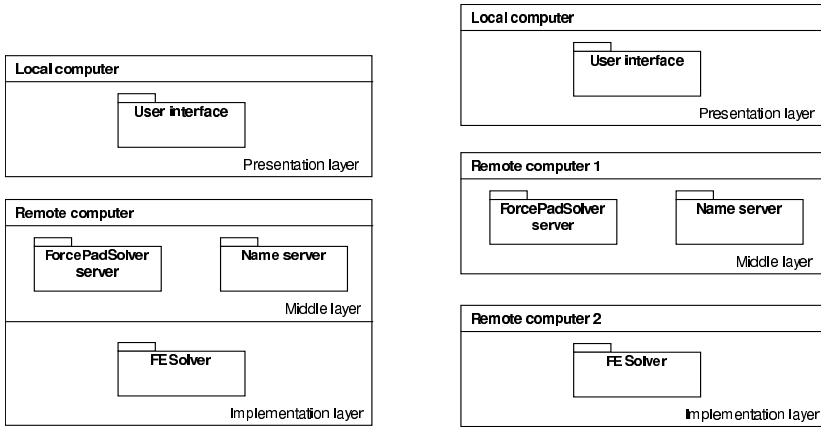
Fig. 16. Remote configuration 1 and 2

with good performance. The 2D graphics in ForcePAD is implemented using OpenGL [15].

One goal of the client application is to hide the CORBA implementation from the user. The user should not be able to notice that the client is using CORBA for interfacing with the ForcePADSolver server.

7 Conclusion

Using a three-tier implementation with interfaces and components, creates a very flexible finite element application. The three-tier implementation protects the client applications from changes in configuration and solver design. By using interfaces when communicating with components, the need to recompile client software when a new functionality is introduced in the solver components is reduced. Interfaces can also be published enabling other software to use the finite element application in an effective way. The CORBA specification also enables new ways of using software. Applications can expose functionality to other applications which can use it either locally or remotely, without having to use intermediate files. By using a special language IDL to define the interfaces to objects and functions, users of the objects can freely choose to implement client applications in any language supported by their CORBA implementation. CORBA exposed objects and methods can also be used in scripting environments to effectively control complex simulations. Client software can easily distribute calculations over available workstations. High Performance Computing (HPC) centres would be able to host a set of applications as CORBA objects. From a web site, users can register themselves as users and download client applications that connect to the objects. This would make high performance computing more available to a wider user group.

References

1. Object Management Group, Inc., http://www.omg.org, 2000

2. Microsoft Corporation, DCOM Technical Overview, 1996

3. M. Dolenc and J. Duhovnik, Designing distributed component-based finite element software, ECPPM 2002

4. B. Spitzak, Fast Light Toolkit FLTK, http://www.fltk.org, 2000

5. N. Frisch and T. Ertl, Embedding Visualisation Software into a Simulation Environment, Proceedings of the Spring Conference on Computer Graphics, Bratislava, 2000

6. Division of Structural Mechanics, Lund Univeristy, ForcePAD, http://www.byggmek.lth.se/bmresources/forcepad, 2001

7. Fnorb the Python CORBA ORB, http://www.fnorb.org, 2002

8. GNOME Computing made easy, http://www.gnome.org, 2002

9. GNU Project, http://www.gnu.org, 2000

10. M. Henning and S. Vinoski, Advanced CORBA Programming with C++, Addison Wesley Longman Inc., 1999

11. R. Larsson, Encapsulation of a finite element program using a distributed object model, Masters dissertation TVSM-5095, Division of Structural Mechanics, Lund University, 1999

12. J. Lindemann, O. Dahlblom and G. Sandberg, An Approach For Distribution Of Resources In Structural Analysis Software, ECCM 99, Munich, Germany, 1999

13. R. Davies, Newmat09: C++ matrix library, http://webnz.com/robert/cpp_lib.htm #newmat09, 2001

14. White Paper on Benchmarking Version 1.0, Object Management Group, December 27, 1999

15. OpenGL, http://www.opengl.org, 2000

16. The Open Group, http://www.opengroup.org/dce, 2000

17. ORBacus 4.1.0, http://www.iona.com/products/orbacus_home.htm, 2002

18. R. Orfali and D. Harkey, Client/server programming with Java and CORBA. - 2nd ed., John Wiley and Sons Inc., 1998

19. Perl Mongers, http://www.perl.org, 2002

20. Simplified Wrapper and Interface Generator, http://www.swig.org, 2002

21. Tcl developer site, http://www.scriptics.com/, 2002

22. Python Language Website, http://www.python.org, 2002

23. Ruby Programmer's Best Friend, http://www.ruby-lang.org/en, 2002

24. Sun Microsystems Inc., JavaTM Remote Method Invocation, http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html, 2001

25. G. Schussel, Client/Server: Past, Present and Future, http://www.dciexpo.com/geos/dbsejava.htm, 1996

26. A. Gokhale and D. Schmidt, Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks, IEEE Transaction on Computers, Volume 47, No. 4, April, 1998

27. T. Forkert, H. Kersken, A. Schreiber, M. Strietzel and K. Wolf, The Distributed Engineering Framework TENT, In proc. of Vector and Parallel Processing - VECPAR 2000, LNCS 1981, pages 38-46, Springer Verlag, 2000

A ForcePAD IDL description

The following section contains the IDL interface description used when generating implementation and client interface code for the ForcePADSolver server.

```
#ifndef _ForcePadSolver_
#define _ForcePadSolver_

#pragma prefix "localdomain.domain"

module ForcePadSolver {

    // Type definitions

    typedef long RetVal;
    typedef sequence<double> TStiffnessVector;
    typedef sequence<double> TDisplVector;
    typedef sequence<double> TResultVector;
    typedef sequence<double> TForceVector;
    typedef sequence<long>   TForceDofVector;
    typedef sequence<double> TBCVector;
    typedef sequence<long>   TBCDofVector;

    // FemGrid interface

    interface FemGrid {
        void setSize(in long rows, in long cols);
        void setStiffness(in TStiffnessVector rowStiffness);
        void setForces(in TForceVector forces, in TForceDofVector forceDofs);
        void setBCs(in TBCVector bcs, in TBCDofVector bcDofs);
        void getDisplacements(out TDisplVector displacements);
        void getResults(out TResultVector results);
        void remove();
    };

    // FemGridFactory interface

    interface FemGridFactory {
        FemGrid create();
    };

    // FemSolver interface

    enum TErrorType {
        ET_NO_ERROR,
        ET_NO_ELEMENTS,
        ET_NO_BCS,
        ET_NO_LOADS,
        ET_UNSTABLE,
        ET_INVALID_MODEL,
        ET_LOAD_OUTSIDE_AE,
        ET_BC_OUTSIDE_AE
    };

    interface FemSolver {
        void execute();
        TErrorType getLastError();
        void remove();
    };

    // FemSolverFactory interface

    interface FemSolverFactory {
        FemSolver create();
    };

    // FemSystem interface
```

```
    interface FemSystem {
        FemGrid getFemGrid();
        FemSolver getFemSolver();
        void remove();
    };

    // FemSystemFactory interface

    interface FemSystemFactory {
        FemSystem create();
    };
};

#endif
```

B   ForcePADSolver server implementation code

This appendix contains the implementation code for the ForcePADSolver server.
The main program initialising the ForcePADSolver server is also listed. Some
code is deliberately left out, indicated with special remarks in the code.

B.1   Header file ForcePADSolver_impl.h

```
#ifndef ___ForcePadSolver_impl_h__
#define ___ForcePadSolver_impl_h__

#include <ForcePadSolver_skel.h>

#include "FemGrid.h"
#include "Forces.h"
#include "BCs.h"

#include <vector>

namespace ForcePadSolver {

class FemGrid_impl : virtual public POA_ForcePadSolver::FemGrid,
                     virtual public PortableServer::RefCountServantBase
{
    FemGrid_impl(const FemGrid_impl&);
    void operator=(const FemGrid_impl&);

    PortableServer::POA_var poa_;
private:
    CFemGrid* m_femGrid;
    CForces* m_forces;
    CBCs* m_bcs;
public:
    FemGrid_impl(PortableServer::POA_ptr);
    ~FemGrid_impl();

    virtual PortableServer::POA_ptr _default_POA();

    virtual void setSize(CORBA::Long rows, CORBA::Long cols)
        throw(CORBA::SystemException);
    virtual void setStiffness(const ForcePadSolver::TStiffnessVector& rowStiffness)
        throw(CORBA::SystemException);
    virtual void setForces(const ForcePadSolver::TForceVector& forces,
                           const ForcePadSolver::TForceDofVector& forceDofs);
    virtual void setBCs(const ForcePadSolver::TBCVector& bcs,
                        const ForcePadSolver::TBCDofVector& bcDofs);
    virtual void getDisplacements(ForcePadSolver::TDisplVector_out displacements)
```

```
        throw(CORBA::SystemException);
    virtual void getResults(ForcePadSolver::TResultVector_out results)
        throw(CORBA::SystemException);
    virtual void remove()
        throw(CORBA::SystemException);

    // Non-CORBA access methods

    CBCs* getBCs();
    CForces* getForces();
    CFemGrid* getFemGrid();
};

class FemGridFactory_impl : virtual public POA_ForcePadSolver::FemGridFactory,
                            virtual public PortableServer::RefCountServantBase
{
    FemGridFactory_impl(const FemGridFactory_impl&);
    void operator=(const FemGridFactory_impl&);

    PortableServer::POA_var poa_;
public:

    FemGridFactory_impl(PortableServer::POA_ptr);
    ~FemGridFactory_impl();

    virtual PortableServer::POA_ptr _default_POA();

    virtual ForcePadSolver::FemGrid_ptr create()
        throw(CORBA::SystemException);
};

class FemSolver_impl : virtual public POA_ForcePadSolver::FemSolver,
                       virtual public PortableServer::RefCountServantBase
{
    FemSolver_impl(const FemSolver_impl&);
    void operator=(const FemSolver_impl&);

    PortableServer::POA_var poa_;
private:
    CFemGrid* m_femGrid;
    double m_maxNodeValue;
    double m_maxStressValue;
    CForces* m_forces;
    CBCs* m_bcs;
    ForcePadSolver::TErrorType m_errorStatus;
public:
    FemSolver_impl(PortableServer::POA_ptr);
    ~FemSolver_impl();

    virtual PortableServer::POA_ptr _default_POA();

    virtual void execute()
        throw(CORBA::SystemException);
    virtual ForcePadSolver::TErrorType getLastError()
        throw(CORBA::SystemException);
    virtual void remove()
        throw(CORBA::SystemException);

    // Non-CORBA access methods

    void setBCs(CBCs* bcs);
    void setForces(CForces* forces);
    void setFemGrid(CFemGrid* femGrid);
};

class FemSolverFactory_impl : virtual public POA_ForcePadSolver::FemSolverFactory,
                              virtual public PortableServer::RefCountServantBase
{
```

```cpp
    FemSolverFactory_impl(const FemSolverFactory_impl&);
    void operator=(const FemSolverFactory_impl&);

    PortableServer::POA_var poa_;

public:

    FemSolverFactory_impl(PortableServer::POA_ptr);
    ~FemSolverFactory_impl();

    virtual PortableServer::POA_ptr _default_POA();

    virtual ForcePadSolver::FemSolver_ptr create()
        throw(CORBA::SystemException);
};

class FemSystem_impl : virtual public POA_ForcePadSolver::FemSystem,
                       virtual public PortableServer::RefCountServantBase
{
    FemSystem_impl(const FemSystem_impl&);
    void operator=(const FemSystem_impl&);

    PortableServer::POA_var poa_;
private:
    ForcePadSolver::FemGrid_ptr     m_femGridRef;
    ForcePadSolver::FemSolver_ptr  m_femSolverRef;
    FemGrid_impl* m_femGridImpl;
    FemSolver_impl* m_femSolverImpl;
public:

    FemSystem_impl(PortableServer::POA_ptr);
    ~FemSystem_impl();

    virtual PortableServer::POA_ptr _default_POA();
    virtual ForcePadSolver::FemGrid_ptr getFemGrid()
        throw(CORBA::SystemException);
    virtual ForcePadSolver::FemSolver_ptr getFemSolver()
        throw(CORBA::SystemException);
    virtual void remove()
        throw(CORBA::SystemException);
};

class FemSystemFactory_impl : virtual public POA_ForcePadSolver::FemSystemFactory,
                              virtual public PortableServer::RefCountServantBase
{
    FemSystemFactory_impl(const FemSystemFactory_impl&);
    void operator=(const FemSystemFactory_impl&);

    PortableServer::POA_var poa_;

public:

    FemSystemFactory_impl(PortableServer::POA_ptr);
    ~FemSystemFactory_impl();

    virtual PortableServer::POA_ptr _default_POA();

    virtual ForcePadSolver::FemSystem_ptr create()
        throw(CORBA::SystemException);
};

} // End of namespace ForcePadSolver

#endif
```

B.2 Implementation file ForcePADSolver_impl.cpp

```cpp
#include <OB/CORBA.h>
#include <ForcePadSolver_impl.h>

#include <iostream>
using namespace std;

#include "calfem.h"
#include <set>

//
// Constructor for the FemGrid implementation
//
ForcePadSolver::FemGrid_impl::FemGrid_impl(
    PortableServer::POA_ptr poa)
    : poa_(PortableServer::POA::_duplicate(poa))
{
    // Construct implementation objects

    m_femGrid = new CFemGrid();
    m_femGrid->setUseImage(false);

    m_forces = new CForces();
    m_bcs = new CBCs();
}

ForcePadSolver::FemGrid_impl::~FemGrid_impl()
{
    // Delete implemenation objects

    delete m_femGrid;
    delete m_forces;
    delete m_bcs;
}

////////////////////////////////////////////////////////////
///// FemGrid_impl access methods                      /////
///// getFemGrid(), getForces(), getBCs(), left out    /////
////////////////////////////////////////////////////////////

PortableServer::POA_ptr
ForcePadSolver::FemGrid_impl::_default_POA()
{
    return PortableServer::POA::_duplicate(poa_);
}

//
// Implements the CORBA method setSize
//
void
ForcePadSolver::FemGrid_impl::setSize(
    CORBA::Long rows,
    CORBA::Long cols)
    throw(CORBA::SystemException)
{
    m_femGrid->setGridSize(rows, cols);
}

//
// Implements the CORBA methods setStiffness
//
void
ForcePadSolver::FemGrid_impl::setStiffness(
    const ForcePadSolver::TStiffnessVector& rowStiffness)
    throw(CORBA::SystemException)
{
    int i, j, k;
```

```
    int rows, cols;

    // Initialise the stiffness grid

    m_femGrid->getGridSize(rows, cols);
    m_femGrid->initGrid();
    m_femGrid->initDofs();
    m_femGrid->initResults();

    // Transfer the incoming rowStiffness vector
    // to the internal implementation grid

    k = 0;

    for (i=0; i<rows; i++)
        for (j=0; j<cols; j++)
        {
            m_femGrid->setGridValue(i, j, 0, (float)rowStiffness[k++]);
            m_femGrid->setGridValue(i, j, 1, (float)rowStiffness[k++]);
        }
}

//
// Implements the CORBA setForces method
//
void
ForcePadSolver::FemGrid_impl::setForces(
    const ForcePadSolver::TForceVector& forces,
    const ForcePadSolver::TForceDofVector& forceDofs)
    throw(CORBA::SystemException)
{
    int i;

    m_forces->clear();

    // Transfer incoming force vectors to internal
    // force list instance

    for (i=0; i<forces.length(); i++)
        m_forces->add(forceDofs[i], forces[i]);
}

//
// Implements the CORBA setBCs method
//
void
ForcePadSolver::FemGrid_impl::setBCs(
    const ForcePadSolver::TBCVector& bcs,
    const ForcePadSolver::TBCDofVector& bcDofs)
    throw(CORBA::SystemException)
{
    int i;

    m_bcs->clear();

    // Transfer incoming bc vectors to internal
    // bc list instance

    for (i=0; i<bcs.length(); i++)
        m_bcs->add(bcDofs[i], bcs[i]);
}

//
// Implements the CORBA getDisplacements method
//
void
ForcePadSolver::FemGrid_impl::getDisplacements(
    ForcePadSolver::TDisplVector_out displacements)
```

```
    throw(CORBA::SystemException)
{
    int i;

    // Instantiate CORBA displacement return vector

    displacements = new ForcePadSolver::TDisplVector;

    // Get displacements from internal implementation

    int displSize = m_femGrid->getDisplacementSize();

    // Resize CORBA vector

    displacements->length(displSize-1);

    // Copy internal results to CORBA return vector

    for (i=0; i<displSize-1; i++)
    {
        (*displacements)[i] = m_femGrid->getDisplacement(i+1);
    }
}

//
// Implements CORBA getResults method
//
void
ForcePadSolver::FemGrid_impl::getResults(
    ForcePadSolver::TResultVector_out results)
    throw(CORBA::SystemException)
{
    // Instantiate CORBA result return vector

    results = new ForcePadSolver::TResultVector;

    int i, j, k, l;
    int rows, cols;
    double values[3];

    // Resize return vector from grid size

    m_femGrid->getGridSize(rows, cols);
    results->length(rows*cols*2*3);

    // Copy results to CORBA result vector

    l = 0;

    for (i=0; i<rows; i++)
        for (j=0; j<cols; j++)
            for (k=0; k<2; k++)
            {
                m_femGrid->getResult(i, j, k, values);
                (*results)[l++] = values[0];
                (*results)[l++] = values[1];
                (*results)[l++] = values[2];
            }
}

//
// Implements CORBA remove method
//
void
ForcePadSolver::FemGrid_impl::remove()
    throw(CORBA::SystemException)
{
    // Get object id of this
```

```
    PortableServer::POA_var poa = _default_POA();
    PortableServer::ObjectId_var id = poa->servant_to_id(this);
    poa->deactivate_object(id);
}


/////////////////////////////////////////////////////////
///// FemGridFactory_impl empty constructor left out    /////
///// FemGridFactory_impl empty destructor left out     /////
///// FemGridFactory_impl::_default_POA() left out      /////
/////////////////////////////////////////////////////////


//
// Implements CORBA FemGridFactory create method
//
ForcePadSolver::FemGrid_ptr
ForcePadSolver::FemGridFactory_impl::create()
    throw(CORBA::SystemException)
{
    // Create a FemCalc implementation Object

    FemGrid_impl* impl = new FemGrid_impl(_default_POA());
    PortableServer::ServantBase_var result = impl;
    return impl->_this();
}


/////////////////////////////////////////////////////////
///// FemGridSolver_impl empty constructor left out     /////
///// FemGridSolver_impl empty destructor left out      /////
/////////////////////////////////////////////////////////


void ForcePadSolver::FemSolver_impl::setFemGrid(CFemGrid *femGrid)
{
    m_femGrid = femGrid;
}


void ForcePadSolver::FemSolver_impl::setForces(CForces *forces)
{
    m_forces = forces;
}


void ForcePadSolver::FemSolver_impl::setBCs(CBCs *bcs)
{
    m_bcs = bcs;
}


/////////////////////////////////////////////////////////
///// FemSolverFactory_impl::_default_POA() left out    /////
/////////////////////////////////////////////////////////


//
// Implements FemSolver CORBA method execute
//
void
ForcePadSolver::FemSolver_impl::execute()
    throw(CORBA::SystemException)
{
    ////////////////////////////////////////
    ///// THIS CODE HAS BEEN LEFT OUT //////
    /////                             //////
    ///// Code contains the solver    //////
    ///// code used in ForcePAD       //////
    ////////////////////////////////////////
}


//
// Implements FemSolver CORBA method last error
//
```

```
ForcePadSolver::TErrorType
ForcePadSolver::FemSolver_impl::getLastError()
    throw(CORBA::SystemException)
{
    // Returns error status from last execution

    ForcePadSolver::TErrorType _r = m_errorStatus;
    return _r;
}

//
// Implements FemSolver CORBA method remove
//
void
ForcePadSolver::FemSolver_impl::remove()
    throw(CORBA::SystemException)
{
    // Get object id of this

    PortableServer::POA_var poa = _default_POA();
    PortableServer::ObjectId_var id = poa->servant_to_id(this);
    poa->deactivate_object(id);
}

///////////////////////////////////////////////////////////
///// FemSolverFactory_impl empty constructor left out  /////
///// FemSolverFactory_impl empty destructor left out   /////
///// FemSolverFactory_impl::_default_POA() left out     /////
///////////////////////////////////////////////////////////

//
// Implements FemSolverFactory CORBA method create
//
ForcePadSolver::FemSolver_ptr
ForcePadSolver::FemSolverFactory_impl::create()
    throw(CORBA::SystemException)
{
    // Create a FemCalc implementation Object

    FemSolver_impl* impl = new FemSolver_impl(_default_POA());
    PortableServer::ServantBase_var result = impl;
    return impl->_this();
}

//
// FemSystem_impl constructor
//
ForcePadSolver::FemSystem_impl::FemSystem_impl(
    PortableServer::POA_ptr poa)
    : poa_(PortableServer::POA::_duplicate(poa))
{
    // Create a node set servant

    m_femGridImpl = new FemGrid_impl(_default_POA());
    m_femSolverImpl = new FemSolver_impl(_default_POA());
    m_femSolverImpl->setFemGrid(m_femGridImpl->getFemGrid());
    m_femSolverImpl->setForces(m_femGridImpl->getForces());
    m_femSolverImpl->setBCs(m_femGridImpl->getBCs());

    // Create a CORBA object reference to FemGrid

    m_femGridRef = m_femGridImpl->_this();
    m_femGridImpl->_remove_ref();

    // Create a CORBA object reference to FemSolver

    m_femSolverRef = m_femSolverImpl->_this();
    m_femSolverImpl->_remove_ref();
```

```
}

////////////////////////////////////////////////////////
///// FemSolverFactory_impl empty destructor left out   /////
///// FemSolverFactory_impl::_default_POA() left out     /////
////////////////////////////////////////////////////////

//
// IDL:byggmek.lth.se/ForcePadSolver/FemSystem/getFemGrid:1.0
//
ForcePadSolver::FemGrid_ptr
ForcePadSolver::FemSystem_impl::getFemGrid()
    throw(CORBA::SystemException)
{
    ForcePadSolver::FemGrid_ptr _r = m_femGridRef;
    return _r;
}

//
// IDL:byggmek.lth.se/ForcePadSolver/FemSystem/getFemSolver:1.0
//
ForcePadSolver::FemSolver_ptr
ForcePadSolver::FemSystem_impl::getFemSolver()
    throw(CORBA::SystemException)
{
    ForcePadSolver::FemSolver_ptr _r = m_femSolverRef;
    return _r;
}

//
// Implements FemSystem CORBA method remove
//
void
ForcePadSolver::FemSystem_impl::remove()
    throw(CORBA::SystemException)
{
    // Remove owned servants

    m_femGridImpl->remove();
    m_femSolverImpl->remove();

    // Get object id of this

    PortableServer::POA_var poa = _default_POA();
    PortableServer::ObjectId_var id = poa->servant_to_id(this);
    poa->deactivate_object(id);
}

////////////////////////////////////////////////////////
///// FemSolverFactory_impl empty constructor left out  /////
///// FemSolverFactory_impl empty destructor left out   /////
///// FemSolverFactory_impl::_default_POA() left out     /////
////////////////////////////////////////////////////////

//
// IDL:byggmek.lth.se/ForcePadSolver/FemSystemFactory/create:1.0
//
ForcePadSolver::FemSystem_ptr
ForcePadSolver::FemSystemFactory_impl::create()
    throw(CORBA::SystemException)
{
    // Create a FemCalc implementation Object

    FemSystem_impl* impl = new FemSystem_impl(_default_POA());
    PortableServer::ServantBase_var result = impl;
    return impl->_this();
}
```

B.3 Server main program

```
#include <OB/CORBA.h>
#include <OB/CosNaming.h>

#include "ForcePadSolver_impl.h"

#include <cstdlib>

#ifdef WIN32
#include <windows.h>
#endif

#include <iostream>
#include <fstream>

namespace std {};
using namespace std;

// Global variable avaiable to console handler

CosNaming::Name name;
CosNaming::NamingContext_var inc;
CosNaming::NamingContext_var dfemc;
CosNaming::NamingContext_var models;
CosNaming::NamingContext_var factories;
CORBA::ORB_var orb;


// Console event handler (WIN32)
//
// Removes all naming information in case of a
// console event

#ifdef WIN32
BOOL WINAPI handler_routine(DWORD dwCtrlType)
{
    //////////////////////////////////////////////////
    //// Code left out                            ////
    //////////////////////////////////////////////////
}
#endif

int main(int argc, char* argv[])
{
    // Handle console window shutdown (WIN32)

#ifdef WIN32
    SetConsoleCtrlHandler(handler_routine, TRUE);
#endif

    // Initialize ORB

    cout << "Initializing orb." << endl;

    orb = CORBA::ORB_init(argc, argv);

    // Get reference to root POA

    cout << "Resolving reference to POA." << endl;

    CORBA::Object_var poaObj =
        orb->resolve_initial_references("RootPOA");

    PortableServer::POA_var poa =
        PortableServer::POA::_narrow(poaObj);

    // Get reference to NameService
```

78

```
cout << "Resolving reference to NameService" << endl;

CORBA::Object_var obj;
try
{
    obj = orb -> resolve_initial_references("NameService");
}
catch(const CORBA::ORB::InvalidName&)
{
    cerr << "  " << argv[0] << ": can't resolve 'NameService'" << endl;
    return 1;
}

inc = CosNaming::NamingContext::_narrow(obj);
assert(!CORBA::is_nil(inc));

// Activate the Root POA manager

cout << "Activating POA manager." << endl;

PortableServer::POAManager_var mgr =
    poa->the_POAManager();

mgr->activate();

// Create a FemSystem server

cout << "Creating factory." << endl;

ForcePadSolver::FemSystemFactory_impl femSystemFactory(poa);
ForcePadSolver::FemSystemFactory_var object = femSystemFactory._this();

// Creating a naming graph

cout << "Adding name information to NameService." << endl;

name.length(1);
name.length(1);
name[0].id = CORBA::string_dup("ForcePad");
dfemc = inc->new_context();
try {
    inc->bind_context(name, dfemc);
} catch (const CosNaming::NamingContext::AlreadyBound &) {
    cout << "  Context ForcePad found." << endl;
}

name.length(2);
name[0].id = CORBA::string_dup("ForcePad");
name[1].id = CORBA::string_dup("Factories");
factories = inc->new_context();
try {
    inc->bind_context(name, factories);
} catch (const CosNaming::NamingContext::AlreadyBound &) {
    cout << "  Context Factories found." << endl;
}

cout << "Binding object to NameService." << endl;

name.length(3);
name[0].id = CORBA::string_dup("ForcePad");
name[1].id = CORBA::string_dup("Factories");
name[2].id = CORBA::string_dup("FemSystemFactory");
inc->rebind(name, object);

// Allow orb to start processing requests.

cout << "Starting orb." << endl << endl;
```

```
    orb->run();

    return 0;

}
```

C   ForcePADSolver client code

This code is from the execute() method in the ForcePAD client application.

```
ForcePadSolver::FemSystem_var femSystem;

try
{
    int i, j, k, l, dofCount;

    // Initialize ORB

    CORBA::ORB_var orb = CORBA::ORB_init(m_argc, m_argv);

    // Get initial naming context

    CORBA::Object_var nameRef;
    try
    {
        nameRef = orb -> resolve_initial_references("NameService");
    }
    catch(const CORBA::ORB::InvalidName&)
    {
        cerr << m_argv[0] << ": can't resolve 'NameService'" << endl;
        throw 0;
    }

    CosNaming::NamingContext_var inc;
    inc = CosNaming::NamingContext::_narrow(nameRef);
    assert(!CORBA::is_nil(inc));

    // Get model factory reference

    CORBA::Object_var systemFactoryRef;
    CosNaming::Name name;
    name.length(3);
    name[0].id = CORBA::string_dup("ForcePad");
    name[1].id = CORBA::string_dup("Factories");
    name[2].id = CORBA::string_dup("FemSystemFactory");

    try {
        systemFactoryRef = inc->resolve(name);
    } catch (const CosNaming::NamingContext::NotFound &) {
        cerr << "No name for FemSystemFactory factory." << endl;
        throw 0;
    } catch (const CORBA::Exception &e) {
        cerr << "Resolve failed: " << e << endl;
        throw 0;
    }

    // Create a FemSystemFactory object

    ForcePadSolver::FemSystemFactory_var femSystemFactory;
    try
    {
        femSystemFactory = ForcePadSolver::FemSystemFactory::_narrow(systemFactoryRef);
    }
    catch (const CORBA::SystemException& se)
```

80

```
{
    cerr << "Cannot narrow FemSystemFactory reference: " << se << endl;
    throw 0;
}

if (CORBA::is_nil(femSystemFactory))
{
    cerr << "femSystemFactory is nil." << endl;
    throw 0;
}

// Create a FemSystem object

femSystem = femSystemFactory->create();

if (CORBA::is_nil(femSystem))
{
    cerr << "FemSystem is nil." << endl;
    throw 0;
}

// Get a FemGrid

ForcePadSolver::FemGrid_var femGrid = femSystem->getFemGrid();

if (CORBA::is_nil(femGrid))
{
    cerr << "FemGrid is nil." << endl;
    throw 0;
}

// Get a FemCalc calculation control object

ForcePadSolver::FemSolver_var femSolver = femSystem->getFemSolver();

if (CORBA::is_nil(femSolver))
{
    cerr << "FemCalc is nil." << endl;
    throw 0;
}

int rows, cols;
int nDof;
int bwLeftRight;
int bwBottomTop;
int maxBandwidth;

// Set gridsize

m_femGrid->getGridSize(rows, cols);
femGrid->setSize(rows, cols);

// Set stiffness

int nStiffness = rows*cols*2;
ForcePadSolver::TStiffnessVector stiffnessVector(nStiffness);
stiffnessVector.length(nStiffness);

l = 0;
float value;

for (i=0; i<rows; i++)
    for (j=0; j<cols; j++)
        for (k=0; k<2; k++)
        {
            value = m_femGrid->getGridValue(i, j, k);
            stiffnessVector[l++] = (double)value;
        }
```

```
femGrid->setStiffness(stiffnessVector);

// Set forces

ForcePadSolver::TForceVector forces;
ForcePadSolver::TForceDofVector forceDofs;

bool loadsDefined = false;
double x, y;
double vx, vy;
int dofs[2];
int forceCount = 0;

int nLoads = m_femGrid->getPointLoadSize();

m_femGrid->clearPoints();

if (nLoads==0)
{
    m_errorStatus = ET_NO_LOADS;
    throw 0;
}

forces.length(nLoads*2);
forceDofs.length(nLoads*2);

CForce* pointLoad = m_femGrid->getFirstPointLoad();

while (pointLoad!=NULL)
{
    pointLoad->getPosition(x, y);
    value = pointLoad->getValue();
    pointLoad->getDirection(vx, vy);
    m_femGrid->getNearestDofs((int)x, (int)y, dofs);

    if (dofs[0]>0)
    {
        forces[forceCount] = vx*value;
        forceDofs[forceCount++] = dofs[0];
        forces[forceCount] = vy*value;
        forceDofs[forceCount++] = dofs[1];

        loadsDefined = true;
    }

    pointLoad = m_femGrid->getNextPointLoad();
}
cout << endl;

if (!loadsDefined)
{
    m_errorStatus = ET_LOAD_OUTSIDE_AE;
    throw 0;
}

femGrid->setForces(forces, forceDofs);

// Define constraints

ForcePadSolver::TBCVector bcs;
ForcePadSolver::TBCDofVector bcDofs;
int nBCs = m_femGrid->getPointConstraintsSize();

if (nBCs==0)
{
    m_errorStatus = ET_NO_BCS;
    throw 0;
```

```
}

int nConstraints = m_femGrid->getPointConstraintsSize();
CConstraint* pointConstraint = m_femGrid->getFirstPointConstraint();

set<int> uniqueDofs;
set<int>::iterator si;
bool bcsDefined = false;

while (pointConstraint!=NULL)
{
    pointConstraint->getPosition(x, y);
    value = pointConstraint->getValue();
    m_femGrid->getNearestDofs((int)x, (int)y, dofs);

    if (dofs[0]>0)
    {
        switch (pointConstraint->getConstraintType()) {
        case CConstraint::CT_XY:
            uniqueDofs.insert(dofs[0]);
            uniqueDofs.insert(dofs[1]);
            bcsDefined = true;
            break;
        case CConstraint::CT_X:
            uniqueDofs.insert(dofs[0]);
            bcsDefined = true;
            break;
        case CConstraint::CT_Y:
            uniqueDofs.insert(dofs[1]);
            bcsDefined = true;
            break;
        default:
            uniqueDofs.insert(dofs[0]);
            uniqueDofs.insert(dofs[1]);
            bcsDefined = true;
            break;
        }
    }

    pointConstraint = m_femGrid->getNextPointConstraint();
}

if (!bcsDefined)
{
    m_errorStatus = ET_BC_OUTSIDE_AE;
    throw 0;
}

// Remove doubly defined dofs

int bcCount = 0;

bcs.length(uniqueDofs.size());
bcDofs.length(uniqueDofs.size());

for (si=uniqueDofs.begin(); si!=uniqueDofs.end(); si++)
{
    int dof = (*si);
    bcs[bcCount] = 0.0;
    bcDofs[bcCount++] = dof;
}

uniqueDofs.clear();

femGrid->setBCs(bcs, bcDofs);

// Solve system
```

```
        ForcePadSolver::TErrorType error;

        femSolver->execute();
        error = femSolver->getLastError();

        // Get displacements

        ForcePadSolver::TDisplVector* displacements;

        femGrid->getDisplacements(displacements);

        m_maxNodeValue = -1.0e300;

        m_femGrid->setDisplacementSize(displacements->length());

        for (i=0; i<displacements->length(); i++)
        {
            m_femGrid->setDisplacement(i+1, (*displacements)[i]);
            if (fabs((*displacements)[i])>m_maxNodeValue)
                m_maxNodeValue = fabs((*displacements)[i]);
        }

        delete displacements;

        m_femGrid->setMaxNodeValue(m_maxNodeValue);

        // Finally destroy all object references

        femSystem->remove();
}
catch (const CORBA::Exception& e)
{
        cerr << "Uncaught CORBA exception: "
            << e << endl;

        try {
            femSystem->remove();
        }
        catch (const CORBA::Exception& e)
        {
            return;
        }
}
```

Paper III

Software for Numerical Simulation of Drying Induced Deformation of Wooden Products

Software for Numerical Simulation of
Drying Induced Deformation of Wooden Products

Ola Dahlblom[1], Jonas Lindemann[1] and Sigurdur Ormarsson[2]

1) Division of Structural Mechanics, LTH, Lund University, Box 118, SE-221 00 Lund, Sweden
2) Department of Structural Engineering and Mechanics, Chalmers University of Technology, SE-412 96 Göteborg, Sweden

ABSTRACT

A previously developed computational model for 3D finite element simulations of wood during moisture changes is in the present work provided with a special-purpose graphical user interface. The software is designed to use a personal computer for the graphical user interface and to have possibility to use distributed computational resources for the simulation.

INTRODUCTION

Distortion of sawn timber due to changes in moisture content is a serious problem. One method to improve the shape stability is to glue wooden pieces together in an optimal way. In the design process of such products, it is necessary to perform computer simulations to predict the distortions. A computer model for 3D finite element simulations of wood deformation during moisture changes has previously been developed (Ormarsson 1999, Dahlblom et al. 2001, Ormarsson et al. 2001). To perform the simulations it is necessary to have detailed information about the moisture and temperature conditions, as well as the mechanical loading. It is also necessary to have a detailed description of the material properties and their variation with moisture content and temperature, as well as with the position within the log. The orientation of the wood fibres must also be considered. In a glued product, the shape stability is to a high extent dependent on the orientation of the pieces.

The previously developed computational model has been successfully applied to different wooden products. To make it suitable for industrial application, a user interface for handling generation of input data and presentation of results is necessary. The interface presented in this paper is under development and the work is sponsored by Swedish wood industry. The software is designed as a Microsoft Windows application using standard menus and toolbars.

USER INTERFACE

General user interface

The application user interface uses a project-based layout, see FIGURE 1. In the left side of the main window, a project view is shown with four folders; Logs and boards, Products, Drying schedules and Sawing patterns. The Logs and boards folder contains a set of logs from which boards can be defined. In the Products folder, laminated products can be created using the individual boards listed in the Logs and boards folder.

The middle part of the application shows the model view, see FIGURE 2. The model view shows different views, depending on what the user has selected in the project view. When an object is selected in the project view, a special property view is shown in the right side of the main window. Depending on what item is selected, the property view can contain one or more tabs.

The user interface is designed to be highly configurable. Toolbars, project views and property windows can be detached and placed floating as separate windows or docked into to the main window at a user specified position. This gives the user the possibility to adjust the user interface to his or her liking. To support advanced users, keyboard shortcuts exist for most menu and toolbar commands, enabling quick and effective usage of the application.
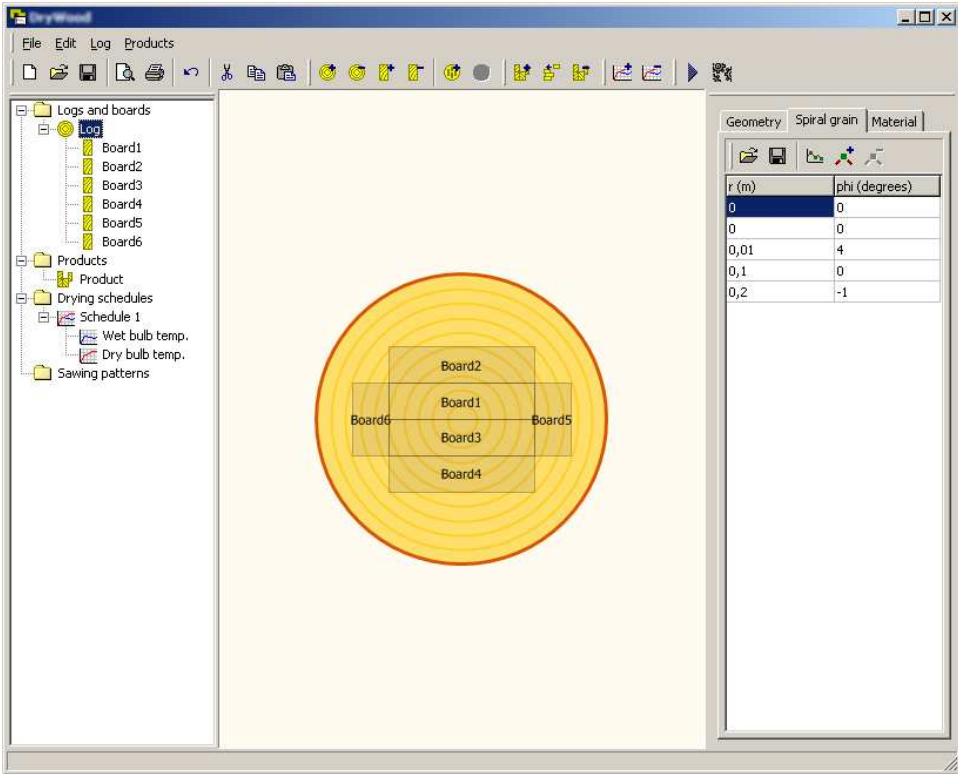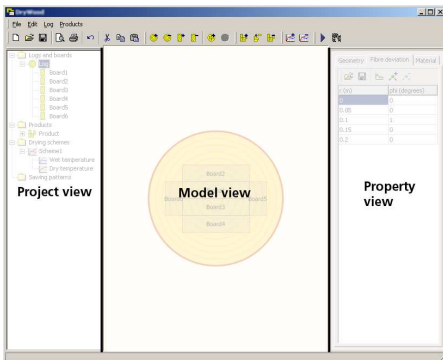
FIGURE 1. Application main window

FIGURE 2. Different areas in the user inerface main window

FIGURE 3. Project view

Project view

The project view displays all objects in the application, see FIGURE 3. In this view objects can be selected, renamed, moved and deleted. When an object is
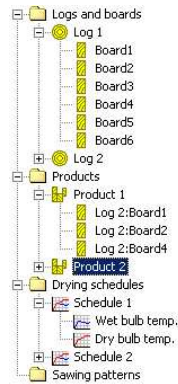
selected in the project view, the model view and property windows are updated accordingly.

To enable quick and easy assembly of products the application extensively uses the notion of drag-and-drop. When a log with a sawing pattern has been cre-

ated, the boards can be dragged onto the products defined in the Product folder, as in FIGURE 4. A product in the application may be a single board or a laminated timber product. When a product is created, it is shown in the project window in the "Products" folder. When boards are added, the product view is also updated to reflect the finished product. Drying schedules are also assigned by dragging them onto the product.
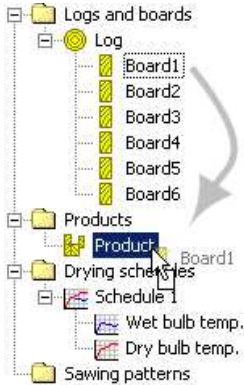
FIGURE 4. Adding boards to products using drag and drop

Model view

The model view shows different views, depending on the currently selected item in the project view. Selecting a log will bring up the log editor and selecting a product will bring up a product editor. In the log editor, boards can be positioned and resized, using the mouse. For more precise updates the position and size of the boards can be updated numerically using the property window.

Changing boards in the log editor will also update the boards in the product editor. FIGURE 5 shows the log editor and FIGURE 6 shows two different laminate products consisting of three boards, as they appear in the product editor
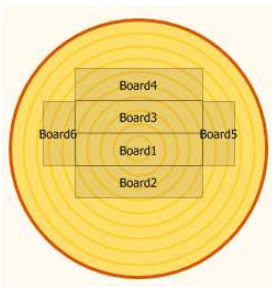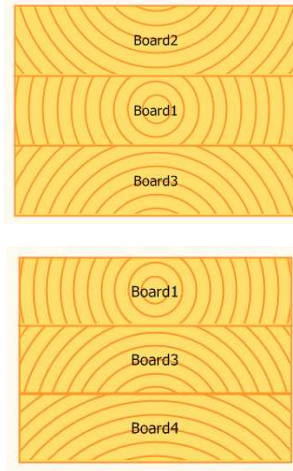

FIGURE 5. Log editor in model view

FIGURE 6. Two different laminated product configurations as they appear in the product editor.

Property window

The property window shows the properties of the item selected in the project view or model view. For example, selecting a log brings up the properties for the log such as butt-end radius and top-end radius, spiral grain and material properties. To use other material parameters for a log than those provided by the application, custom materials can be created. If an object has many properties, they are divided into categories and displayed under tabs in the property window as shown in FIGURE 7.
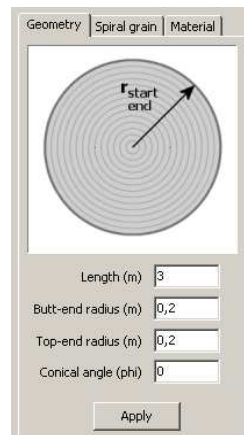

FIGURE 7. Property view (log selected)

Spiral grain angle can be edited using a spreadsheet table or using a special envelope editor where the points

can be dragged visually using the mouse. The spiral grain angle editor is shown in FIGURE 8.
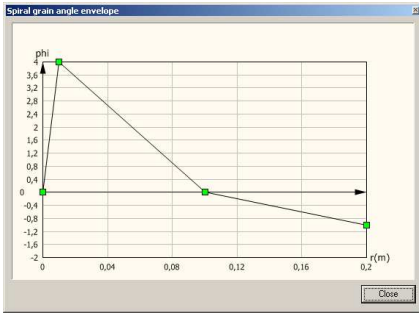

FIGURE 8. Spiral grain angle editor

Handling of drying schedules

To simulate the drying process, drying schedules are needed. Schedules can be created either by entering them in a special table, importing them from a spreadsheet application, or by visually dragging points in an envelope editor as shown in FIGURE 9. All schedules are stored in a special folder called "Drying schedules" in the project view, see FIGURE 10. To use a drying schedule, it can be dragged onto a product in the same way as boards are dragged onto the products.
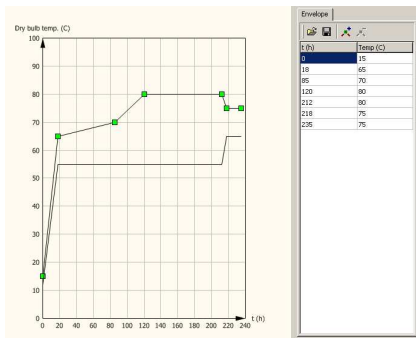

FIGURE 9. Drying schedule editor

FIGURE 10. Drying schedules in the project view

Simulation control

Because simulations can be time consuming it is important to let the user control all aspects of the simulation jobs. In the application, the user can submit mul-

tiple jobs that can be run in the background on the local machine or remotely on a different machine. This functionality is controlled using the job manager, see FIGURE 11. Status information is continuously updated in the job manager, to display progress and error information. The user can also terminate any erroneous jobs submitted.
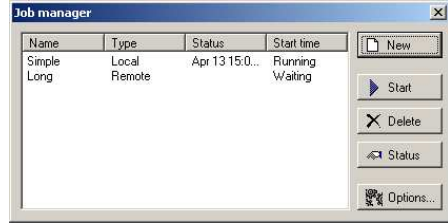

FIGURE 11. Job manager

Result view

The result view displays results from the simulations. Results that can be displayed are: displacement history, stresses and strains. The simulation is time dependant, so the result view has a special slider bar for changing the current simulation step. To handle fast drawing of results the result view uses OpenGL (OpenGL 2003). The viewpoint of the result view can be updated in real-time by moving the mouse using a "virtual trackball". It is also possible to change the magnification factors and colour scales in the result view with the associated property window. As an example of results, distortions of the two products according to FIGURE 6 are shown in FIGURE 12.
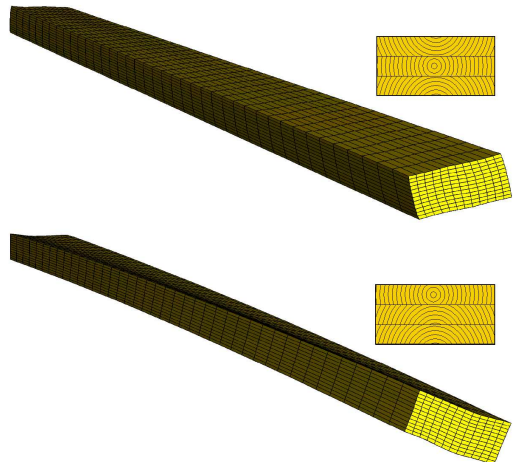

FIGURE 12. Distortion of the laminated products according to FIGURE 6.

IMPLEMENTATION

User interface

The user interface is designed around a set of components defining the functionality. This reduces dependencies and complexity making the code easier to maintain. All components use a special class library defining the different items in the workspace. The software is developed using Borland Delphi (Borland Delphi 2003), a rapid application development tool (RAD), which enables the user interface to be designed effectively using visual tools.

To enable easy integration width other software, the application stores its data using XML (W3C 2003) files. An XML file is a standardised way of storing structured information using tags similar to those found in HTML-files. XML-files are normal text files editable in a text editor.

Modular solver design

To facilitate using different solver solutions the application implements a special modular solver design, see FIGURE 13. Instead of communicating directly with the finite element solvers, the software interacts with solvers using a set of plug-ins. Each plug-in implement the same interface, enabling dynamic loading of different solver solutions. The responsibility of the plug-in is to generate necessary input files to the solver, execute and monitor the solver process and finally to process and handle the solver output files. The plug-in architecture also enables the user interface to be easily configured to take advantage of distributed resources using CORBA or GRID middleware, as described in Lindemann (1999, 2001, 2002, 2003), Foster (2001) and Larsson (1999).
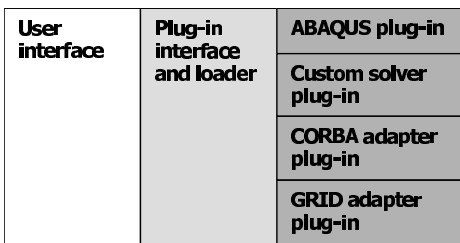
User interface	Plug-in interface and loader	ABAQUS plug-in
		Custom solver plug-in
		CORBA adapter plug-in
		GRID adapter plug-in

FIGURE 13. Application modular solver architecture

CONCLUSION

In many cases research results are presented in reports or computational codes which may be difficult to directly use in industry. The present application is an effort to encapsulate a computational code developed in a research project into an application that can be used by people in the industry. The developed application enables users to easily assemble wooden products that are to be studied. Effects of different drying schedules can be studied. Different ways of laminating together boards in products can also easily be studied.

The modular design of the solver implementation will also facilitate distributing the simulation code to resources located at physically different locations. In the future there will also be possibilities to access resources on the computational GRID.

REFERENCES

Borland Delphi 2003: http://www.borland.com

CORBA 2003: http://www.omg.org

Dahlblom O., Petersson H. and Ormarsson S. 2001: Full 3-D FEM-Simulations of Drying Distortions in Spruce Boards based on Experimental Studies, 7th International IUFRO Wood Drying Conference, Tsukuba, Japan, July 9-13

Foster I., Kesselman. C., Tuecke S. 2001: The Anatomy of the Grid - Enabling Scalable Virtual Organizations, to appear: Intl J. Supercomputer Applications

Larsson R. 1999: Encapsulation of a finite element program using a distributed object model, Report TVSM-5095, Structural Mechanics, Lund University

Lindemann J., Dahlblom O., Sandberg G. 1999: An Approach For Distribution Of Resources In Structural Analysis Software, ECCM '99, Munich, Germany

Lindemann J. 2001: Programming and Visualisation Techniques in finite element software, Report TVSM-3050, Structural Mechanics, Lund University

Lindemann J., Sandberg G. and Dahlblom O. 2002: Using CORBA Middleware in Finite Element Software, Lecture Notes in Computer Science 2331, Computational Science - ICCS 2002, 701-710

Lindemann J., Dahlblom O., Sandberg G. 2003: Using CORBA middleware in finite element software, Future Generation Computer Systems, Accepted for publication

OpenGL 2003: http://www.opengl.org, OpenGL - High Performance 2D/3D Graphics

Ormarsson S. 1999: Numerical analysis of moisture-related distortions in sawn timber, PhD Thesis, Department of Structural Mechanics, Chalmers University of Technology, Göteborg

Ormarsson S., Petersson H., Eriksson J. and Dahlblom O. 2001: Improved shape stability of timber products obtained by use of a numerical simulation technique, 7th International IUFRO Wood Drying Conference, Tsukuba, Japan, July 9-13

W3C 2003: Extensible Markup Language (XML), http://www.w3.org/XML

Paper IV

Real-time Visualisation of Fibre Networks

The Visual Computer, 2002

Real-time visualisation
of fibre networks

J. Lindemann,
O. Dahlblom

Division of Structural Mechanics, Lund University,
John Ericssons väg 1, 221 00 Lund, Sweden
E-mail: strucmech@byggmek.lth.se

Different methods of real-time fibre-network visualisation have been studied. Using an extrusion-based method yields very good results, but for large networks the frame rate becomes unacceptably low. To increase the number of fibres that can be visualised in real time, a textured billboard method has been implemented. With this method, an average performance gain of 60% has been achieved, using an OpenGL implementation.

**Key words:** Fibre network – Visualisation – Billboard – Extrusion – Real time

1 Introduction

A fibre network is a material that has a structure consisting of fibres, such as fluff and fibre insulation material. A computer program simulating deformation and fracture in fibre networks has been developed by Heyden [2]. When three-dimensional fibre networks are simulated, the numerical results are often difficult to interpret. To be able to analyse the time-dependent results, different methods have been used to visualise the data. Commercial post-processors often have difficulties dealing with simulations that do not exactly fit into the finite element concept of visualisation. When the initial work on visualising fibre networks started, the commercial visualisation system NAG Explorer [5] was used, with disappointing results. The NAG explorer is a very powerful tool when dealing with structured and unstructured grids, but it does not handle the unusual geometry of fibre networks well. This paper presents a texture-based method of visualising fibre-network simulations in real time. The method has been tested for simulations ranging from 900 to 2500 fibres.

2 Visualising fibres, using an extrusion-based method

Fibres can be visualised in many different ways. A straightforward approach is to sweep a section along the spine of the fibre in a piecewise linear manner. To achieve a realistic tubular fibre with this method, a section with at least six sides has to be used, see Fig. 1.

The extrusion method was first implemented using VRML97 [9]. This approach worked well due to the flexibility of the VRML97 specification. The fibres were represented using the extrusion nodes and the connection points as sphere primitives. Time-dependent information could also be represented by using coordinate and colour interpolators. Using extrusion nodes meant that many triangles had to be used to represent the fibres, limiting the number of fibres that could be represented in real time. The interpolators were not as effective as expected. The performance of the VRML97 visualisation largely depends on the viewer used. By using the Cosmo-Player 2.1 plugin [1] on an SGI Onyx2 machine [7], a fibre network consisting of approximately 900 fibres could be rendered in real time at a rate of only 4–5 frames per second. To increase the performance of the extrusion-based method, it was implemented using C++ and OpenGL [6]. The rendering of the
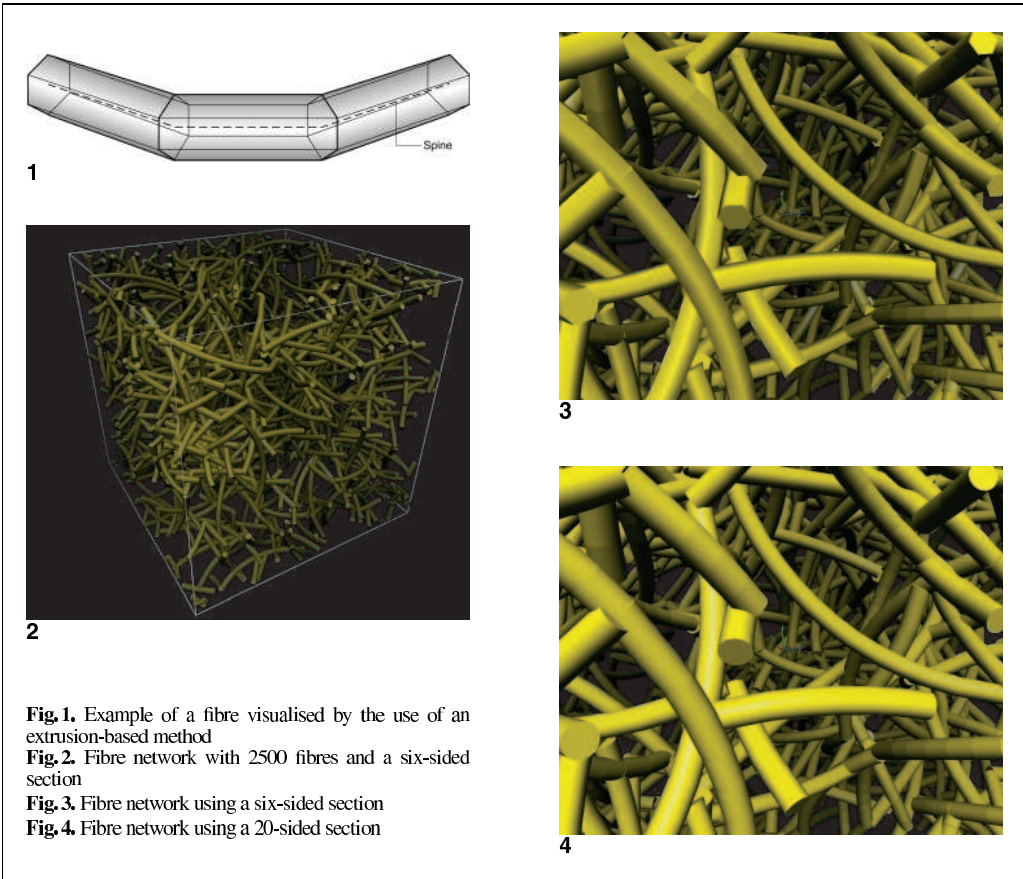
Fig. 1. Example of a fibre visualised by the use of an extrusion-based method

Fig. 2. Fibre network with 2500 fibres and a six-sided section

Fig. 3. Fibre network using a six-sided section

Fig. 4. Fibre network using a 20-sided section

extrusions is performed using the GLE Tubing and Extrusion library [8]. This library is widely used when rendering extruded objects, and has a good performance. The application implemented in C++ achieves a frame rate of around 6 frames per second on an SGI Onyx2 system for a network of 2500 fibres. Figure 2 shows a screen capture of this network.

The main advantage of using this method is image quality, which is increased by adding sides to the fibre section. The drawback is of course a decreased real-time performance. Using more than 12 sides will only enhance the rendering of specular highlights and does not add to the general image quality. Figures 3 and 4 show the network rendered, using 6 and 20 sides in the sections respectively. The vertex interpolation is much smoother when more sides are used, but the six-sided section gives an acceptable quality for real-time use.

3 Visualising fibres, using texture bands

The proposed method uses a single band of triangles with a gradient texture, to give an illusion of a circular section (see Fig. 5). Using this method, the number of triangles only increases by 2 per segment, which is much less than for the extrusion-based method. The problem when drawing just these

96
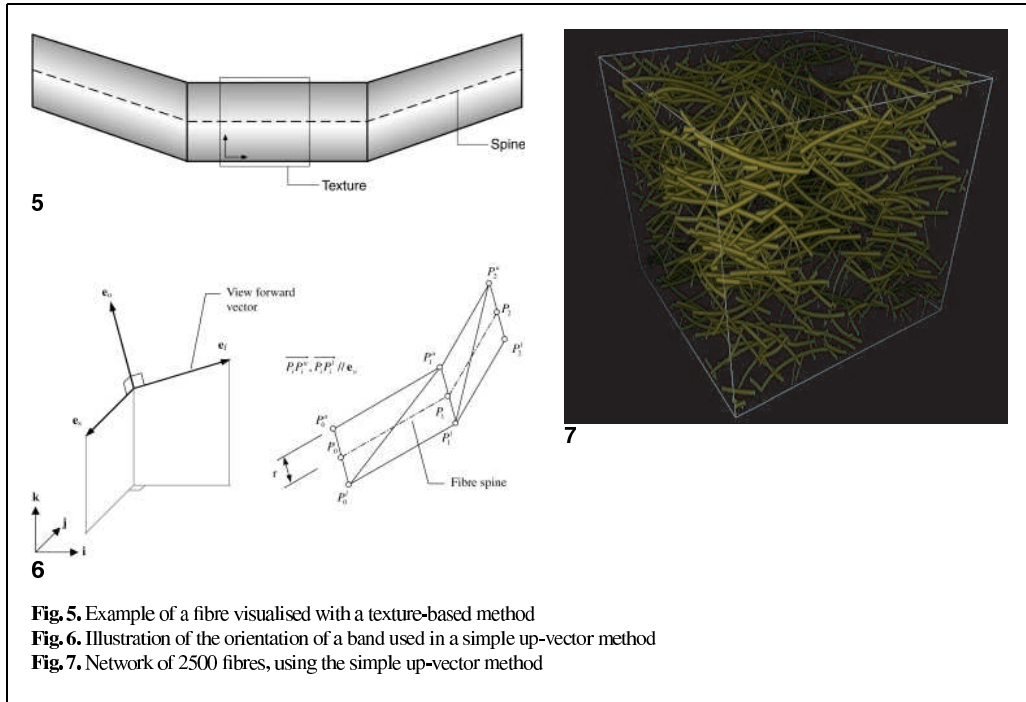
Fig. 5. Example of a fibre visualised with a texture-based method
Fig. 6. Illustration of the orientation of a band used in a simple up-vector method
Fig. 7. Network of 2500 fibres, using the simple up-vector method

bands is that the rendered image largely depends on the viewing angle. At certain angles, the entire network may disappear. To avoid this, the orientation of each band representing a fibre is chosen to depend on the view transformation in such a way that the flat side is always turned towards the viewer. Two methods of view alignment have been implemented.

3.1 Simple up-vector alignment method

In this method, the up-vector of the view transformation is used to create an upper and a lower offset of the original line. The segment is then drawn with triangles between these lines (see Fig. 6). The vector e_f is the normalized view vector determined by the camera position (x_p, y_p, z_p) and the target (x_t, y_t, z_t):

$$e_f = \frac{(x_t - x_p)\boldsymbol{i} + (y_t - y_p)\boldsymbol{j} + (z_t - z_p)\boldsymbol{k}}{\sqrt{(x_t - x_p)^2 + (y_t - y_p)^2 + (z_t - z_p)^2}}. \tag{1}$$

By using this vector, the following vectors are obtained:

$$e_s = \frac{\boldsymbol{j} \times e_f}{|\boldsymbol{j} \times e_f|}, \tag{2}$$

$$e_o = e_f \times e_s, \tag{3}$$

where e_o is perpendicular to the view forward vector e_f and in the same plane as the up vector $\boldsymbol{j}$ and e_f. By using the e_o vector, the points needed to draw the band can be determined by

$$\overrightarrow{OP_i^u} = \overrightarrow{OP_i} + re_o, \tag{4}$$

$$\overrightarrow{OP_i^l} = \overrightarrow{OP_i} - re_o, \tag{5}$$

where P_i is a point on the fibre segment and O is the origin.

Due to its simple implementation, this method is very efficient. The drawback is that fibres oriented in the camera up direction can be difficult to see. The use of this method is illustrated in Fig. 7. Looking at the network in Figs. 7 and 8 at close range reveals the banded nature of the fibres. Even though the band structure can be visibly observed at close
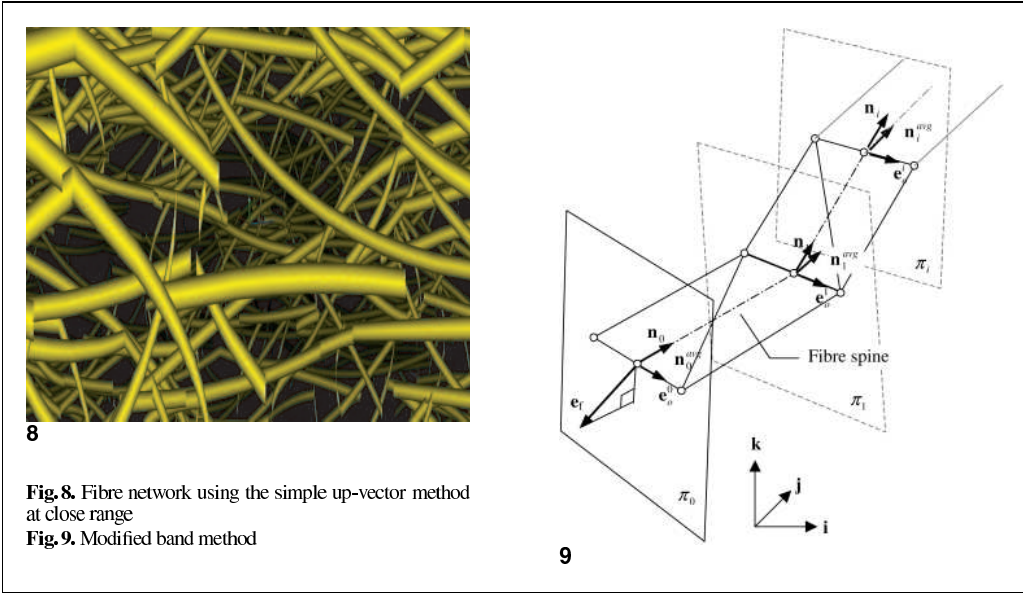
Fig. 8. Fibre network using the simple up-vector method at close range
Fig. 9. Modified band method

range, it is difficult to see at the global level. In studies of network deformations the global level is often of primary interest, and in such situations the method may therefore be useful in spite of its disadvantages at close range.

3.2 Billboard method

To solve the visibility problem with the previous method, a billboard method has been developed. In this method, the fibre band is aligned with the view around each vertex point, and the created points are on the plane with the normal, according to the direction of the spine (see Fig. 9). First the vertex normal vector n_i is calculated from the fibre segment,

$$n_i = \frac{(x_{i+1} - x_i)i + (y_{i+1} - y_i)j + (z_{i+1} - z_i)k}{\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 + (z_{i+1} - z_i)^2}}. \quad (6)$$

An average vertex normal is calculated using

$$n_i^{avg} = \frac{n_{i+1} + n_i}{|n_{i+1} + n_i|}, \quad (7)$$

with the exception of the end-points, where it is given by $n_0^{avg} = n_0$ and $n_n^{avg} = n_{n-1}$. The fibre orientation vector e_0^i can be calculated as

$$e_o^i = \frac{e_f \times n_i^{avg}}{|e_f \times n_i^{avg}|}, \quad (8)$$

where e_f is the normalized view vector defined in (1). By using the e_o vector, the points needed to draw the band can be determined by

$$\overrightarrow{OP_i^u} = \overrightarrow{OP_i} + re_o^i, \quad (9)$$

$$\overrightarrow{OP_i^l} = \overrightarrow{OP_i} - re_o^i, \quad (10)$$

where P_i is a point on the fibre segment.

The billboard method in Figs. 10 and 11 shows a close resemblance to the image obtained with the extrusion method, shown previously in Figs. 3 and 4. The most noticeable difference is the fibre end-points. With the extrusion method, the end caps are represented correctly. Due to the banded geometry used in the textured-band method, the end caps cannot be represented correctly. Another difference between the methods is that in the textured-band method a light source cannot be used.

Using textures also brings other advantages. A texture is a graphical image, and it is therefore possible to use actual images of fibres, to make the network appear more realistic. Figure 12 shows a close-up of a network using a "bumpy" texture, which gives the fibres a more organic look.
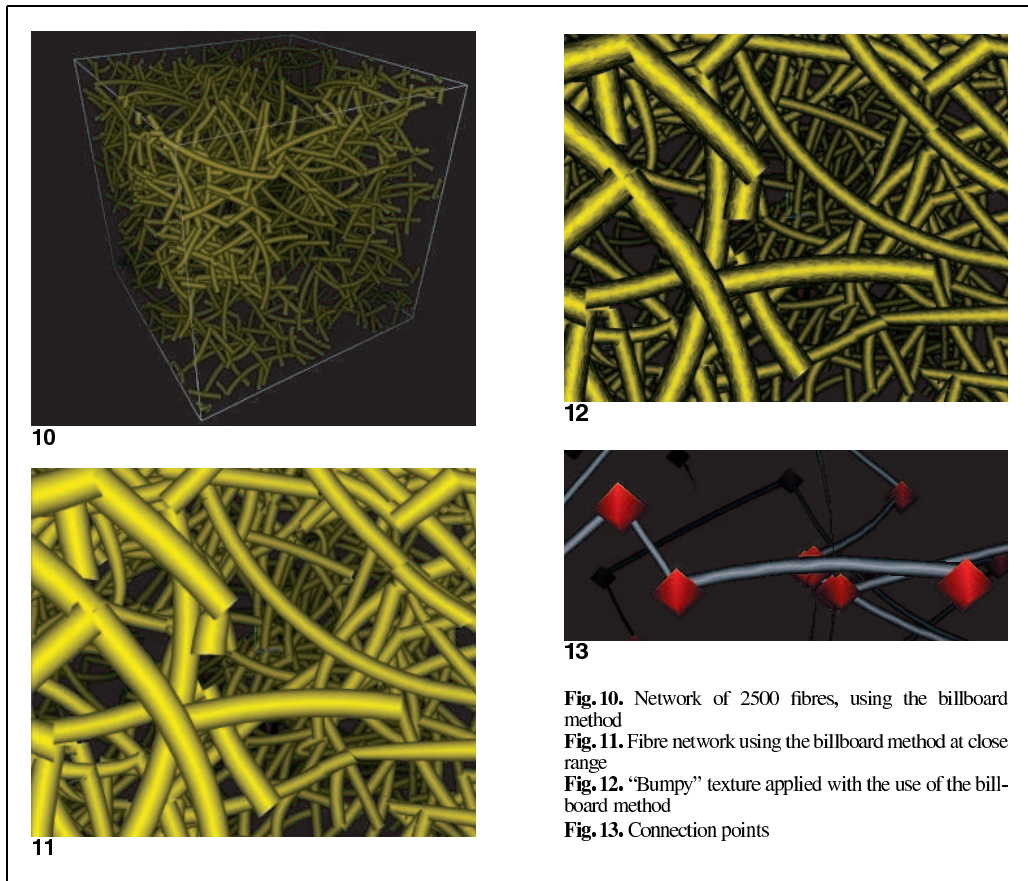
Fig. 10. Network of 2500 fibres, using the billboard method
Fig. 11. Fibre network using the billboard method at close range
Fig. 12. "Bumpy" texture applied with the use of the billboard method
Fig. 13. Connection points

4 Visualising connection-point usage and breakage

A fibre-network model is constructed of a large number of fibres connected with connection points. When the network is subjected to loading, these connection points will become stressed and eventually break. To visualise these connection points, simple spheres are used. Figure 13 shows typical connection points.

The simulation model [2] provides information about connection-point usage. Usage is converted to a colour through a scale and then applied to the sphere representation. To visualise the connection-point usage, it is important to choose a good, intuitive colour scale. A high usage could, for example, be represented by a white colour and low usage by a dark red colour. Because of the complexity of the network, connection points are easily hidden among the many fibres in the model. To remedy this, the fibres should be made dimmer and smaller when connection points are to be studied. To further illustrate a high usage, connection points with a usage above, say, 95% can be enlarged to highlight this aspect in the model. This gives a "popping" effect when viewed through the timesteps. Effective colour maps are shown in Fig. 14. In the same way as enlarging the usage above 95%, sound can be used to forewarn that a connection is about to be broken. The
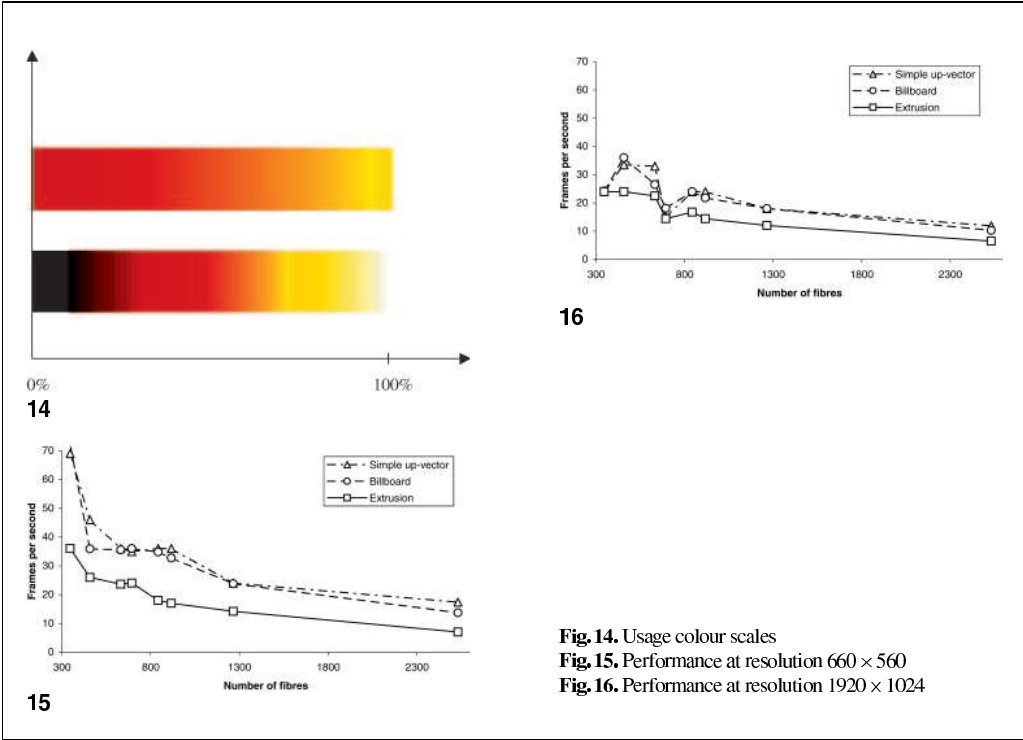
Fig. 14. Usage colour scales
Fig. 15. Performance at resolution 660×560
Fig. 16. Performance at resolution 1920×1024

amount of audible "popping" gives hints on breakages in the model.

When a connection point is broken, according to the simulation model [2], the sphere should be made invisible. A sphere is made invisible by setting the OpenGL alpha value to zero. If this is not done, broken connection points will float around in the model and possibly hide important information.

5 Comparisons

The performance of the methods was evaluated on an SGI Onyx2 machine, by measuring the frame rate when animating the displacement history of fibre networks consisting of different numbers of fibres. An idle loop was used to drive the animation. The performance was measured at full resolution, 1920×1024, and at a reduced resolution, 660×560. Figs. 15 and 16 show the performance achieved.

Performance gain compared with the extrusion method with six-sided sections is shown in Table 1. The difference in performance between a small and a large resolution is due to the fact that the 3D hardware has to render larger triangles at a higher resolution, which decreases the performance. Using

Table 1. Performance increase using a texture-based method (L = large resolution, S = small resolution)

Fibres	Simple S.	Billb. S	Simple L.	Billb. L
347	1.92	2.00	1.00	1.01
458	1.77	1.38	1.40	1.50
632	1.53	1.51	1.47	1.18
694	1.46	1.50	1.04	1.26
843	2.00	1.93	1.43	1.43
917	2.12	1.92	1.67	1.51
1265	1.69	1.68	1.50	1.50
2531	2.49	1.96	1.83	1.58
Average	1.87	1.73	1.42	1.37
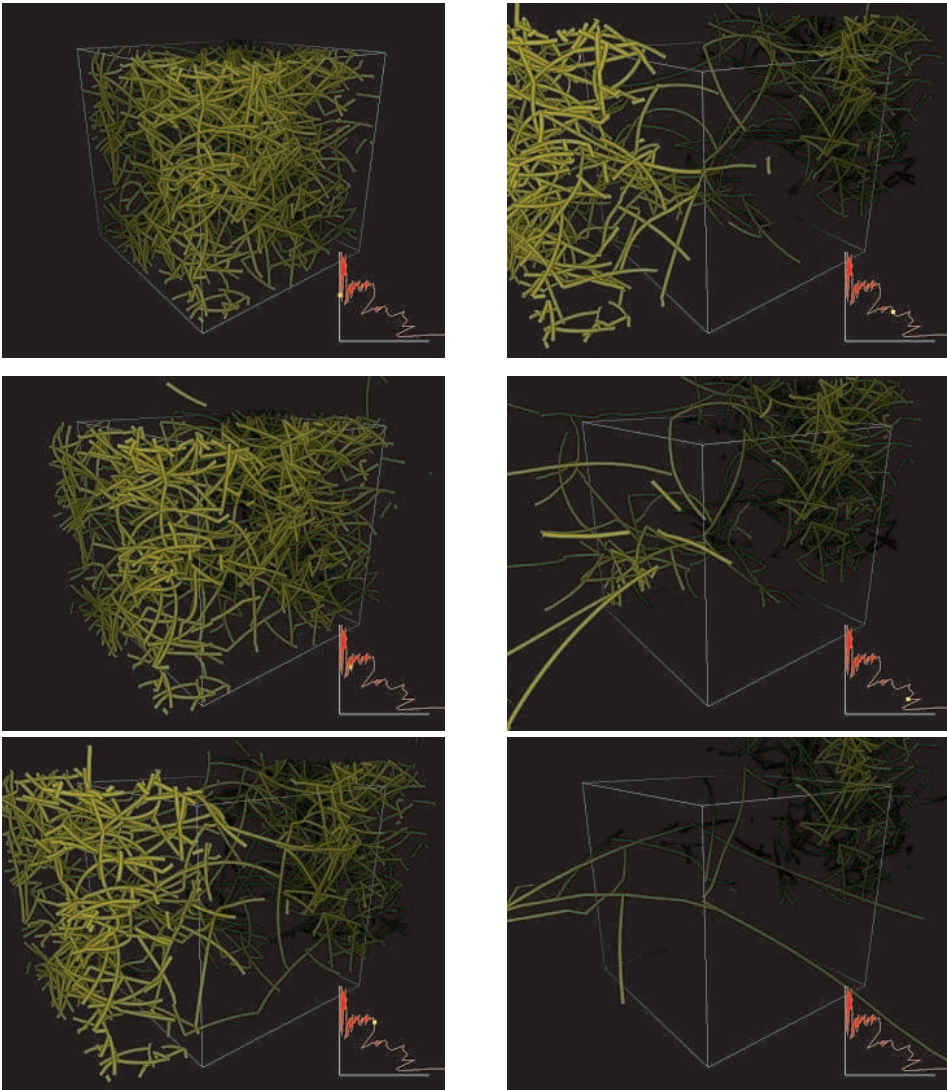
Fig. 17. Snapshots from a simulation

more graphic pipelines in the SGI Onyx2 machine will probably increase the performance when rendering at a higher resolution. Higher texture-based performance of the methods could also be achieved if optimising the band drawing code to the same level as the GLE library [8]. Tests have also been made on a standard 3D accelerated PC. These tests show an even greater performance difference between the extrusion-based method and the banded texture methods.

6 Example of use

Figure 17 shows a sequence of snapshots at selected points from a fibre network simulation of a $2 \times 2 \times 2$ mm cube of cellulose fibre fluff, performed by Heyden [2]. The fibres were arranged in a random structure according to random fibre orientation distribution and the bonds were modelled as non-linear coupling elements representing stick-slip performance. The structure is subjected to a forced displacement in one direction. Network geometry is periodic, such that opposite sides match. This allows the studied cube to be regarded as one of many cubes making up a global structure. The visualisation below consists of 2531 fibres and 331 simulation steps. The diagram in the lower part of the images shows the current position of the simulation in the stress–strain relationship.

7 Implementation

To be able to study the time-dependent behaviour of fibre networks, real-time performance is important. To implement this in real time, OpenGL was chosen, because it is a platform-independent specification for hardware-accelerated 3D equipment. The Ivf++ library [4] was chosen as an abstraction layer on top of OpenGL. Ivf++ is a simple class library, encapsulating OpenGL and GLUT [3] functionality. The main Ivf++ library consists of about 50 classes, among them primitive shapes, selection, level of detail (LOD) and view management. In the texture method, fast switching between timesteps is important. Each fibre stores an initial geometry description and a set of relative displacements for each fibre and timestep. By using this scheme, scaling of displacements can be done efficiently without re-computing all fibre coordinates. Animating the time history is then just a matter of updating the current timestep. Fibre rendering is performed using a specially developed fibre class. This class implements both the extrusion-based method of fibre rendering and the texture-based method. The extrusion-based method is implemented with the GLE library. The textured method is implemented using the OpenGL GL_TRIANGLES primitive. Connection points are rendered in the same way as the fibres, but with the use of a sphere primitive.

8 Conclusions

It has been shown that a texture-based method can be effectively used to visualise large fibre networks in real time with a good visual appearance. The texture-based method can also be effectively used to visualise networks on smaller hardware. In the fibre networks studied, ranging from 900 to 2500 fibres and from 80 to 300 timesteps, all fibres and timesteps could be loaded into memory (SGI Onyx2 with 1 Gb of memory). Typical memory usage for these networks range from 40 Mb to 380 Mb. Using the above approach, real-time update of the timesteps could be made with up to 2500 fibres.

References

1. Cosmo Software (2000) CosmoPlayer 2.1.
   http://www.cai.com/cosmo/
2. Heyden S (2000) A 3D network model for evaluation of mechanical properties of cellulose fibre fluff. Report TVSM-1011, Division of Structural Mechanics, Lund University
3. Kilgard M (2000) The OpenGL Utility Toolkit (GLUT) Programming Interface API version 3.
   http://reality.sgi.com/opengl/spec3/spec3.html
4. Lindemann J (2001) Interactive Visualisation Framework – user's guide. Report TVSM-3038, Division of Structural Mechanics, Lund University
5. NAG (2000) IRIS Explorer. The Numerical Algorithms Group Ltd,
   http://www.nag.co.uk/Welcome_IEC.html
6. OpenGL (2000) http://www.opengl.org
7. Silicon Graphics (2000) Silicon Graphics Onyx2.
   http://www.sgi.com/onyx2
8. Vepstas L (2000) GLE Tubing and Extrusion library.
   http://linas.org/gle/index.html
9. Web3D Consortium (2000) VRML97.
   http://www.web3d.org

Photographs of the authors and their biographies are given on the next page.

JONAS LINDEMANN was born in Sweden in 1970. He received his MS degree in civil engineering in 1997. He is currently working as a PhD student at the Division of Structural Mechanics at Lund University. His research interests include visualisation of finite element simulations, methods for distributing finite element systems using CORBA and DCOM, development of user interfaces for computational codes and educational tools in structural mechanics.

OLA DAHLBLOM received his PhD degree in structural mechanics at Lund University in 1987. He is currently an associate professor at the Division of Structural Mechanics at Lund University. His research activities are concerned with constitutive modelling and finite element simulation of structural behaviour. An important part of the work is development of computer code for computation and for results presentation. Examples of areas of application are simulation of deformation development in sawn wood during moisture content changes and stress development in concrete structures during hardening.

Paper V

An Approach to Teaching Architectural and Engineering Students Utilizing Computational Mechanics Software ForcePAD

AN APPROACH TO TEACHING ARCHITECTURAL AND ENGINEERING STUDENTS UTILIZING COMPUTATIONAL MECHANICS SOFTWARE FORCEPAD

J. Lindemann, Postgraduate student,
Division of Structural Mechanics, Lund University, Lund;
jonas.lindemann@byggmek.lth.se, http://www.byggmek.lth.se

G. Sandberg, Professor,
Division of Structural Mechanics, Lund University, Lund;
goran@byggmek.lth.se, http://www.byggmek.lth.se

K. Olsson, Lecturer
Structural Design, School of Architecture, Chalmers University of Technology, Gothenburg
kg@arch.chalmers.se, http://www.arch.chalmers.se

SUMMARY: *The paper discusses how courses in mechanics can be taught to architectural students in a manner aimed at presenting concepts in such a way that mechanics becomes a inspiration for the design process rather than a limitation to it. It also presents an ideal tool for basic introduction in mechanics to students in civil engineering. In the courses of this sort that have been held, emphasis has been placed on the use of software for facilitating an intuitive understanding of physical matters related to mechanics and how that understanding can be transformed into design sketches. ForcePAD is a comprehensible software for making sketches and investigating patterns in mechanics. Its aim is to enhance the conception of such factors as balance, weight, stability, rest and movement, support forces, stress fields, and deformation. The paper is based on experience with classes of this sort taught both at Lund University and at Chalmers University in Gothenburg (Olsson, 2003), the weekly tasks students have been given in courses of this type being discussed. The pedagogical idea utilized in these classes taught has influenced in the design of the software ForcePAD.*

KEYWORDS: *Teaching, Mechanics, Software, ForcePAD.*

1. INTRODUCTION

Design is inventing. It is about finding shapes that not only have an interesting appearance but also actually are able to contain certain tangible properties. Designing force-carrying structures is about inventing too. A structure that integrates strong spatial qualities with effective use of its material is second to none. (Think about the gothic cathedrals from eight hundred years ago. The delicate stone ribbons embrace the interior space like fan-shaped laces. They are true minimal structures that should be compared to the most efficient aeroplane construction and yet they allow their space to penetrated by the light from the sun and the sky and allow their space to be cooled by natural ventilation through the open ribbons.)

Teaching mechanics to design students is about providing the students with an ability to go from investigating properties to articulation of expression, from inner structure to outer contour. The tool for this endeavour includes simple sketching, fast response to investigate properties of mechanics such as contour deformation, forces and force fields.

Within industrial design and architecture, the structural properties arrived at in the buildings and objects produced are often a consequence primarily of artistic intuition, of strict topology or use of readymade solutions. Their function however are often to carry a load. The aim of teaching should be to make the pattern of abstract forces involved both inspiring and readily accessible to the student of design so that the structure, rather than simply being a functional necessity, provides an image of forces.

Courses in mechanics have the immediate practical goal of providing the understanding and the tools needed for the designing of structures. However, courses should also be taught in such a way that they become a source of inspiration in matters pertaining to design. Although the conceptions of mechanics are abstract, they relate to our

understanding of how constructions form a well-functional structural system. It is also one of the great strengths of mechanics that both the conception and context exits in physical shapes, mechanics allowing us to experiment with materials and shapes so as to create the basis for an intuitive interpretation of the abstract content of the conceptions. We readily understand what is heavy, light, and stable, in equilibrium or seem to be out of balance, or when the structure seems to be at the boundary of what it can withstand in term of exterior forces. The abstract, absolute thinking of science is related to our intuitive understanding since it takes as its reference the world around us, which we can observe and interpret in everyday life. Since the abstract ideas of mechanics exist in the form of physical shapes or are related to these, students can be trained to use them as sources of inspiration in design tasks and in preliminary sketches for these.

2. A WEEKLY TASK – A TRIPOD

In the following, a task students were given is described. Brief accounts of discussions with students are presented to indicate how the processes referred to above influenced them and allowed them to use elements of mechanics as a source of inspiration.

One task given to first-year students of industrial design at Lund University is described for students as follows:

"Three points are defined in a horizontal plane and forms an equal sided triangle. The side length is 1400 mm. A fourth point is defined 600 mm above this plane. At this position in space, you shall be able to place an item weighing at least than 5 kg. The size of the base area of item is 100′100 mm2. The material is corrugated paperboard in sheets of the size 1000′600 mm2. You can assemble the material by using glue, staples, or by knitting. However, you have to choose only one of these means. The support of the structure must be within a circle with the diameter of 100 mm. The structure will be judged by the way it expresses how the load is carried, including how the load is transmitted from loading position to the support positions in the corners of the triangle. The structure should be as light as possible and a volley ball should be able to roll under it."

The major question for the student to consider here is how the solution arrived at expresses the external load and the path of the internal load, i.e. how the visible structure reflects the stresses present in the material, and how the material is utilized to accommodate this stresses. Students are to make sketches of the design of the structure intended. They are also to present arguments in support of their solution. To illustrate this, consider how two of the students presented and developed their arguments. Their brief sketches are presented in Fig. 1 below. The drawing to the left, (a), represents their first suggestion. The basic idea was to have the loading position encircled by the structure. Since large parts of the structure do not contribute to the load-bearing capacity, one can ask whether it is possible both to let their intention of encircling the load be fulfilled and to let each part contribute to supporting the load. Allowing the structural parts to meet above the loading position would be one solution.
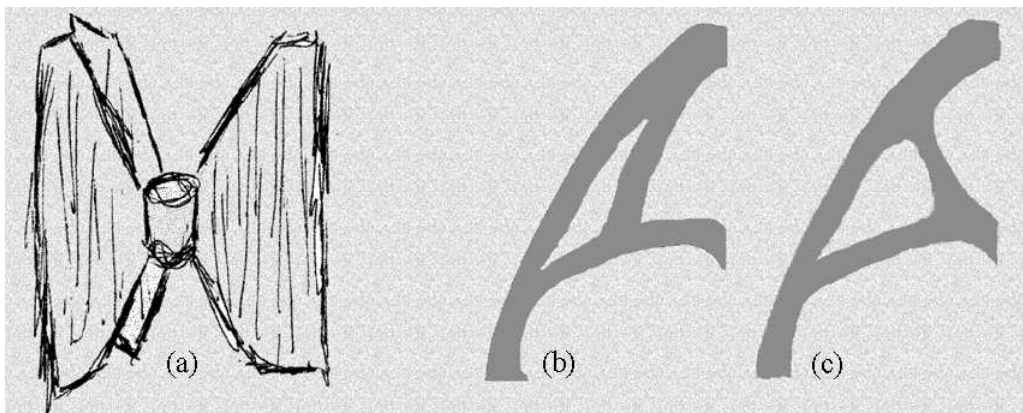

FIG. 1 - The first suggestion for the tripod the structure should encircle the load, (a). Two suggestions for one leg of the tripod, (b)-(c)

(b) shows their first proposal along this line, in connection with which they argued for letting all the structural elements have the same visual direction. It is not evident, however, that this is favourable from a structural point of view. Their third proposal is shown to the right, (c). Here the change shows the flow of forces in the external load instead, interest being directed at where the load is placed. Quick simulations also indicate the flow of the internal forces here to be different.
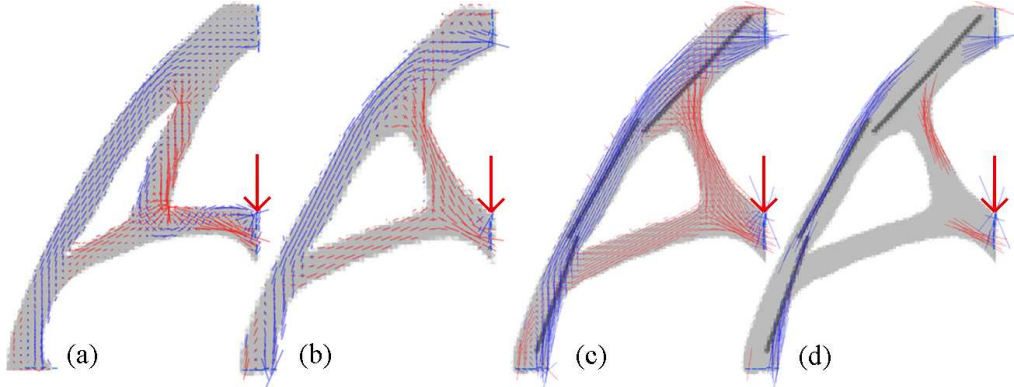
FIG. 2 - Stresses in one of the legs of the tripod, blue indicates compression and red indicates tension stresses, (a) first attempt, (b) final solution, (c) stiffeners have been attached, and (d) only high levels of stress are shown.

Whereas the proposal at the left in Figure 2, (a), gives rise to an unclear and mixed stress field, parts both tensile and compressive states, the proposal (b) is better coordinated. Two parts are both exposed to tensile forces, which of course is favourable (remember, the tripod is constructed of corrugated cardboard). The compression evident in the long element needs to deal with by use of additional stiffeners (c-d), which were introduced in the final solution.

The tripod ready for testing is shown in Fig. 3. Loads were applied until the structure collapsed. This particular tripod yielded with grace under the ultimate load, its rotating downwards as the legs collapsed. Even after the collapse the solution selected looks interesting, since the failure tells such a clear history, see Figure 4.

FIG. 3 - The final solution for a load-carrying tripod.

FIG. 4 - Loading of the structure (left) and the structure as it appeared following failure (right).

One can conclude, on the basis of the results of these brief tasks, that the experiment the students conducted and their discussion of it made them aware of the qualities of the material involved and how these can be used to express and articulate the shape of the structure created and to design and link together its various parts. These qualities are not readily apparent without a tool to make them visible.

3. DESIGN AND IMPLEMENTATION OF FORCEPAD

ForcePAD was designed as an intuitive sketch tool for designers, archtitects and people without the theoretical knowledge of material, shape and force relationships. To be easy to use for inexperienced users, the user interface is based loosely on the design commonly used in standard image drawing programs.

The underlying implementation of ForcePAD is a 2D finite element model based on simple triangular plane stress elements, with 2 degrees of freedom at each node. Users don't create elements directly, but first "paint" stiffness on a screen image. When a calculation is done, the image is transferred to an element grid, which is then solved. The stiffness values for each element in the element grid are determined by calculating the amount of pixels covering the element. Fig. 1 illustrates this.
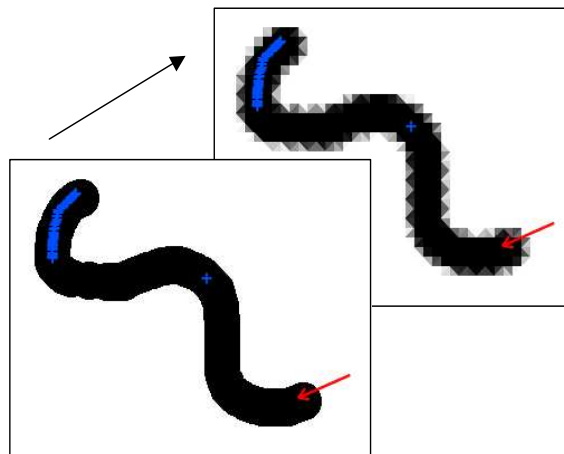

FIGURE 1 - Converting a pixel based image to a finite element grid

3.1 User interface

The ForcePAD system is designed as a direct manipulation system, which according to (Preece, 1994) should have the following properties: Visibility of the objects of interest, rapid and reversible, incremental actions, and replacement of complex command language syntax by direct manipulation of the object of interest.
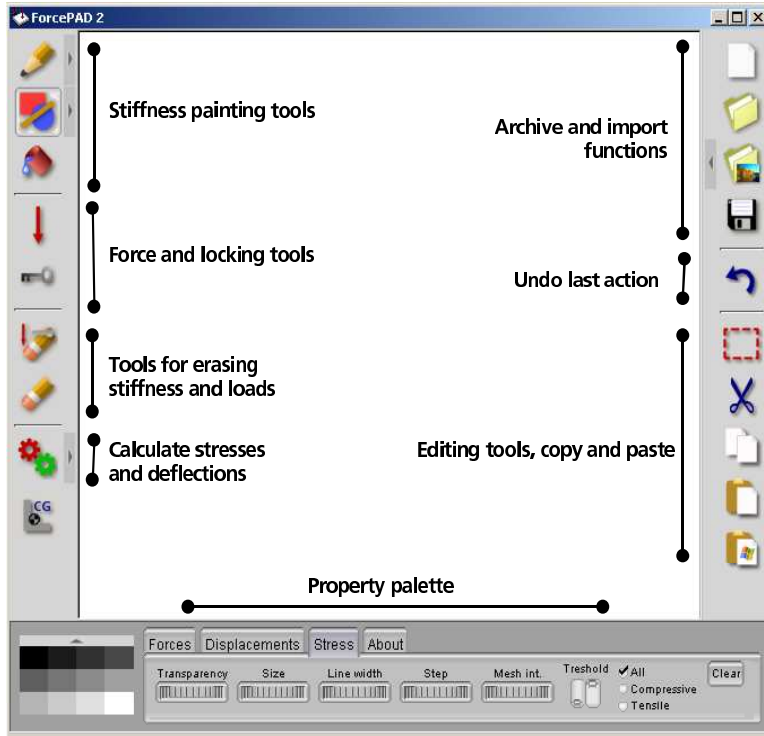

FIGURE 2 - ForcePAD user interface

The user interface of ForcePAD is divided into three main areas: drawing area, toolbars and the property palette. The drawing area is used to paint the structures to be studied and to visualise the displacements and stresses. Often used tools are found in the left toolbar. In the right toolbar, functions for archiving, undo and copy/paste functionality are found. Visualisation properties are found in the property palette.

A problem with many applications today is that most functionality and features are visible in different toolbars and menus, making the user interface very complex. According to (Norman, 1998) there are two ways to overcome this "featurism". First is avoidance or great restrain. Avoidance and restraint is not always possible, because many features are an essential part of an application and cannot be removed. The second way is organisation or modularisation. In this method the application functions are divided into a number of modules each with a limited number of functions. In this way, all features are still present but are represented in small parts that are more comprehensible.

To make ForcePAD easy to use and yet have more features for experienced users, more advanced features are "hidden" using property buttons and "flip-out" toolbars. The property buttons are shown beside the normal button. When a property button is pressed, a small window is shown beside the button with the more advanced features. Fig. 3 illustrates this. Geometric painting functions are also hidden by placing them on a "flip-out" toolbar, which is shown when pressing the button for geometric painting functions.

FIGURE 3 - Property buttons

The icon design in ForcePAD is influenced by the guidelines found in (Shneidermann, 1998) chapter 6.4. The icons in ForcePAD are 36x36 pixels, so that a high colored three-dimensional image can be represented. All icons also have a drop-shadow to make the more clearly identifiable.

3.1.1 "Painting stiffness"

Creating structures in ForcePAD is done by "painting" stiffness. Black represents max stiffness and white no stiffness. Stiffness is chosen by using the stiffness palette. This corresponds to selecting a different color in a normal image editing application.

Painting is done by using one of the three different painting tools: brush, geometric shapes and fill. These correspond to the common tools also found in imaging editing applications.

3.1.2 Creating loads and boundary conditions

Loads in ForcePAD are represented using red arrows. The user creates a load by selecting the load tool and clicking on the point where the load is to be applied. The Direction of the load is set by moving the mouse around the application point. When the mouse is release the force is added to the model.

The term boundary condition is not used in ForcePAD, instead the term locking is used. This is done to make it easier for novice users to understand the concept of boundary conditions. Three kinds of locks are used, locked in x direction indicated by a vertical blue line (|), locked in y direction indicated by a horisontal blue line (-) and locked in x and y direction indicated by a blue cross (+). The choice of line direction has been done emulate a frictionless surface on which the structure is free to move.

3.2 Visualisation

If a tool should be able to be used effectively as a design tool, the visualisation of results from finite element simulation should be an integral part of the design process. To facilitate this the user can control the parameters of the visualisation using the property palette.

FIGURE 4 - Tweakable visualisation controls

Visualising stresses in ForcePAD is done using coloured arrows representing principal stresses. An arrow is coloured red if positive principal stress and blue if negative. Important parameters for controlling the appearance of the stress visualisation are: Transparency of stress arrows, Size of stress arrows and Width of stress arrows. Examples of stress visualisation are shown in fig. 5.
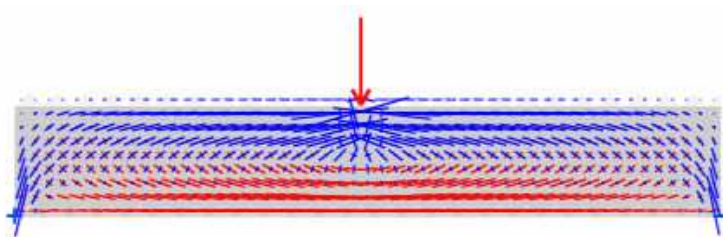
FIGURE 5 - Stress visualisation

Displacements can also be visualised in ForcePAD. The deflection scalefactor can be scaled in real-time using the slider found in the displacements tab in the property palette. Fig. 6 shows a beam deflected by a single force.

FIGURE 6 - Deflected beam structure subjected to a single load

3.3 Software implementation

ForcePAD is implemented in C++. The user interface is created using the Fast Light Toolkit (Spitzak, 2003). This is a lightweight user interface toolkit written in C++. The toolkit can be used on Windows 95/98/NT and most Unix:es with good performance.

The solver is implemented using the newmat09 (Davies, 2002) matrix library. Using this library a finite element solver was implemented using a notation similar to MATLAB (The MathWorks, Inc, 2003) just by changing some of the notation.

Visualisation of the element grid, stress distribution and displacements where implemented using OpenGL (OpenGL, 2003). Using OpenGL good performance is achieved for 2D graphics in both software emulated graphics and hardware supported graphics.

ForcePAD is implemented in a way that it does not change or modify any settings in computer when installed. This also enables it to be installed and executed from any folder, making easier to install it on a network share. The ForcePAD install is also small, ~1.3Mb making it possible to download it over slow Internet connections as well.

4. FORCEPAD ON THE WEB

To support ForcePAD when used in an educational setting, a special web page has been setup. (http://www.byggmek.lth.se → Resources → ForcePAD) At this web page users and student using ForcePAD can download the lastest version, look at video tutorials and ask questions and discuss problem in a electronic web forum. Making the software and documentation available over the Internet make it easier to support.
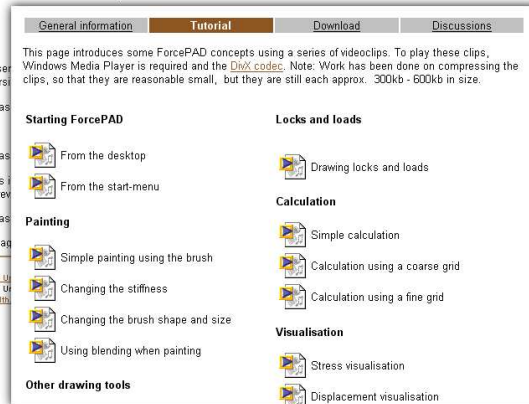
FIGURE 7 - ForcePAD web page

5. CONCLUDING REMARKS

ForcePAD is an educational software programme developed at the Division of Structural Mechanics at Lund University in collaboration with the Division of Building Design at Chalmers. Although it was conceived for use by students of industrial design and architecture, we believe it can be useful for other categories of students as well, due to its unique features. ForcePAD deals with a variety of different matters of physical character within the area of mechanics, such as the centre of gravity, loads, support reactions, deformation, and internal stresses.

A unique feature of it is its simple interface, which clearly mirrors the physical constituents involved. The interface mimics the conditions that sketching on a sheet of paper represent. The immediate consequences that adding material or a line or removing material by scratching has, in terms of changes in form, adds to the simplicity of working with it, allowing ForcePAD to become an intimate part of design sketching in an educational context. Our experience with it is that it supports in a very genuine way a reflective process on the part of the user, providing both insight and inspiration in forming materials into shapes for creative and constructive ends. The programme supports an iterative process of reflective optimisation that the user is guided through, rather than its being software for simply an automatic optimization of shapes. Despite its not being software for advanced mechanical analysis, hidden within it are in fact some advanced finite element tools having optimizing characteristics computationally.

6. REFERENCES

Davies R. (2002). Newmat C++ matrix library, http://www.robertnz.net/nm_intro.htm

Norman D. (1988). The design of everyday things, First Doubleday/Currency

Olsson K.G. (2003). Course homepage for Building Design at Chalmers, http://www.arch.chalmers.se/tema/form-teknik/byggnadskonstruktion/kla1_vt2003/kla1_03.html

OpenGL. (2003). OpenGL - The Industry's Foundation for High Performance Graphics, http://www.opengl.org

Preece J. et al (1994). Human-Computer Interaction, Addison-Wesley

Sandberg G. and Olsson K. and Lindemann J and Lund M. (2002). Images of forces, Proceedings of the, DRS 2002 International Conference, Common Ground, 5 - 8 September 2002

Shneiderman B. (1998). Designing the user interface : strategies for effective human-computer interaction, Third edition, Addison-Wesley

Spitzak B. et al (2003). The Fast Light Toolkit, http://www.fltk.org

The MathWorks, Inc. (2003). MATLAB, http://www.mathworks.com

Paper VI

ObjectiveFrame - An Educational Tool for understanding the Behaviour of Structures

117

ObjectiveFrame - An educational tool for understanding the behaviour of structures

J. Lindemann, O. Dahlblom and G. Sandberg

Division of Structural Mechanics, Lund University
John Ericssons väg 1
221 00 Lund
+46 46 222 73 70
struchmech@byggmek.lth.se

ABSTRACT

To understand the behaviour of structures subjected to loads the 3d beam analysis program ObjectiveFrame has been developed. One of the features of this program is that it can visualise the response of a structure to a user-defined load in real-time. This makes it suitable for use as an educational tool in design science, architecture and structural mechanics. ObjectiveFrame also implements a heads-up display used for displaying messages and toolbars. Most operations in ObjectiveFrame are implemented using a direct manipulation paradigm.

KEYWORDS

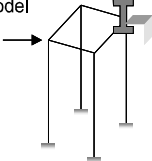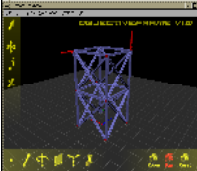Virtual Reality, Real-time visualisation, Structural Mechanics

INTRODUCTION

Creating a 3D beam model is a complex task in most pre-processors, due to the complexity of the user interface. These programs are often designed as general pre-processors handling all kinds of finite element models. ObjectiveFrame was designed to solve some of these problems, creating an intuitive tool for experiments with structures and forces.

To enable users to experiment with a structure, ObjectiveFrame has the ability to visualise the response of structures in real-time, when applied a user-defined load. To make ObjectiveFrame intuitive and easy to use, most operations in the user interface are implemented using a direct manipulation paradigm, giving the users direct feedback of actions taken.

CONCEPTUAL MODEL

A conceptual model is a mental representation of how an object works. An application user interface must convey the conceptual model to the user. ObjectiveFrame uses a conceptual model based on Christiansson [1], see Figure 1.

Domain	Product	Process
Application	Beam model	Edit, change and move nodes, beams, loads, boundary conditions and materials.
System		User interface, editing functions, 3D graphics, view transformations, FE solver.
User	Students, Architects, Engineers	Visualise behaviour of structures

Figure 1 - Conceptual model

In this approach the conceptual model is divided into three domains, the application, the system and user domains. The application domain is the problem area. The system domain is a description of the system. The user domain describes the users and the problems the users want to solve.

The application model is a 3D beam model using 6 degrees of freedom at each node, based on a Bernoulli beam theory.

Due to the low complexity of the user interface in ObjectiveFrame, it has been successfully used in the teaching of mechanics to architectural students. In this context the main user goals are:

- Provide an easy way of constructing a 3D beam model.
- Solving 3D beam problems
- Understanding the behaviour of structures and forces.

According to Preece [7], a direct manipulation system can have the following properties:

- Visibility of the objects of interest
- Rapid and reversible incremental actions
- Replacement of complex command language syntax by direct manipulation of the object of interest.

The system model of ObjectiveFrame was designed as a direct manipulation system with using the goals described above. Reversible action has however not been implemented yet.

ObjectiveFrame also uses the Model-View-Controller paradigm described in Dix [2]. In this paradigm the user interface is managed by three components:

- The model component, representing the application.
- One or more view components, responsible for displaying views of the model.
- The controller component, receiving input from input devices controlling the view and model components.

Figure 2 illustrates this paradigm.
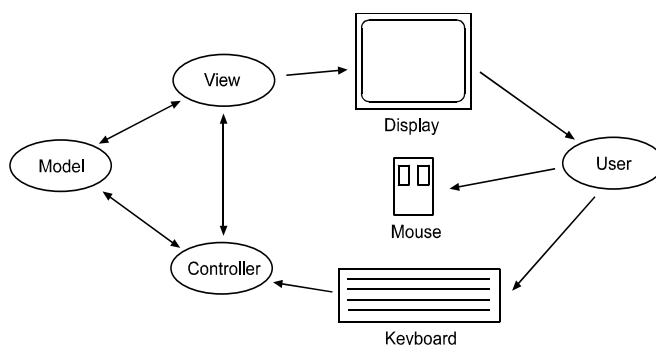
Figure 2 - Model-View-Controller paradigm

ObjectiveFrame only uses one 3d perspective view of the beam model. The design principle used was to give the user the impression of directly manipulating a real 3d beam model. To further the experience, the beam model is displayed shaded using a lightsource.

USER INTERFACE

The main user interface in ObjectiveFrame is the workspace. The workspace is the "workbench" on which the beam models are assembled and consists of an axis and a transparent grid. The user can change the workspace size to fit the model constructed. Object placement, movement and creation are done using a special 3d cursor. The position of the 3d cursor is determined by mapping the mouse coordinates on to the 3d grid. Movement perpendicular to the grid is done by holding down the [ctrl] button. 3d cursor movement in a VR system would use some kind of 3d pointing device instead of the mouse. Figure 3 shows the workspace layout.

Figure 3 - ObjectiveFrame workspace and toolbars

Many modern applications provide toolbars for most user actions. The use of toolbars tends to reduce the available work area of the application. In ObjectiveFrame toolbars have been made transparent, so that the 3d workspace can be seen through them. To make it easier to find different toolbars coloured gradients are also placed behind the toolbar icons. When the mouse is over a toolbar icon, this icon magnified and its colour changed to white. A hint is also displayed in the middle of the screen. The toolbars are also visible in Figure 3.

The technique described is often found in many modern computer games, for example Half-Life [3].

ObjectiveFrame uses seven modes:

- Select
- Move
- Feedback
- Create nodes
- Create beams
- View/zoom
- View/pan

Clicking on a 3d toolbar icon switches between nodes. The current mode is indicated by a red square around the toolbar icon.

In the view modes the user can freely move around the beam model. Using the left mouse button always rotates the view. The right button is used to zoom or pan depending on the current mode.

In the selection mode objects can be selected. A selectable object is highlighted in white, when the mouse is over it. An object is added to the list of selected objects by clicking on it. Clicking on the workspace clears the selection. Selected objects can be moved, deleted, assigned loads, boundary conditions or material properties.
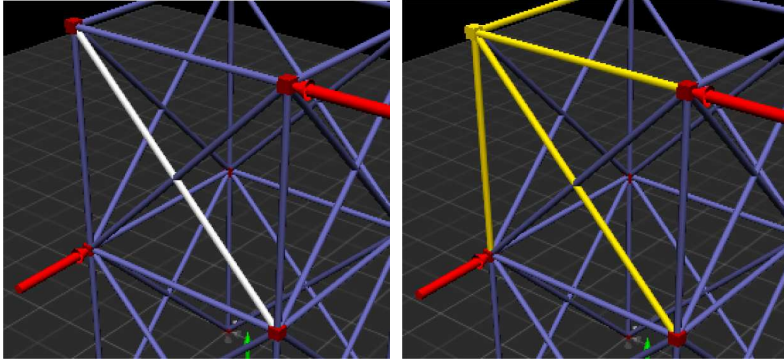

Figure 4 - Highlighting and selection

VISUALISING STRUCTURE RESPONSE

To enable experimentation with the interaction of forces and structure, ObjectiveFrame implements a special feedback mode. In this mode a user-defined load can be applied to the structure. When the user moves the load, the corresponding deflections are computed in real-time. This enables the user to "feel" the response of a structure.
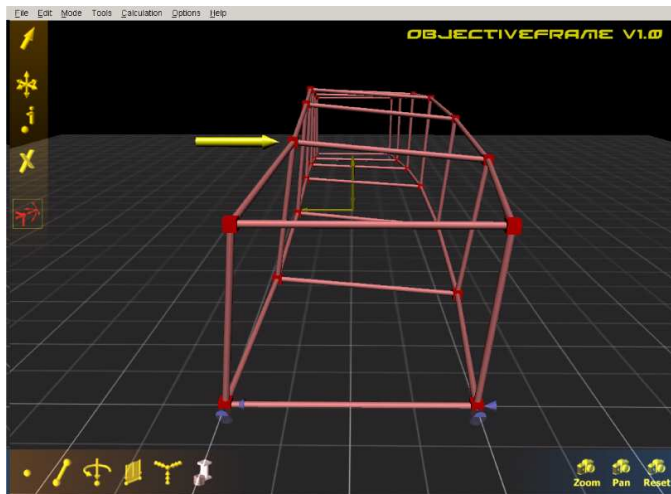
Figure 5 - Structure in feedback mode

When the feedback mode is selected, a node must be selected for the user-defined force. Moving the mouse over a node displays a force over it, indicating that it can be placed at the node. The force is attached to the node by clicking on it. A first calculation of the model is done as well. The force can now be moved using the mouse, updating the structure deflection continuously. Figure 6 shows how the mouse

movements map to the 3d world. In a VR system the force could be controlled directly using some kind of 3d input device.
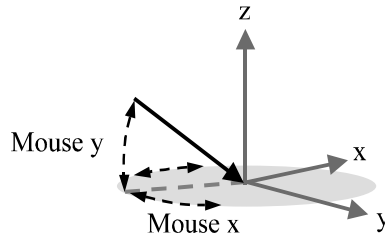
Figure 6 – Force movement

EXAMPLE OF USE

ObjectiveFrame has successfully been used in the course "High Structures - A Creative Investigation" [4] at the School of Architecture at Chalmers University of Technology. In the course the students investigated the architectural possibilities and limitations of high structures. ObjectiveFrame was used as a tool for understanding the effects of different methods for stiffening a structure. Using the feedback method, the students could experiment and evaluate the different methods in real-time on the screen.

Figure 7 - A student group working on the assignment

IMPLEMENTATION

The application was implemented in C++. The 2d user interface parts where implemented using the platform independent toolkit FLTK [8]. The 3d user interface

was implemented using the Interactive Visualisation Framework - Ivf++ [5], which is a set of C++ libraries for creating 3d graphics. This library uses OpenGL [6] as a rasterisation interface.

CONCLUSIONS

It has been shown that a direct manipulation interface can be effectively used in 3d beam analysis. Instant feedback of actions gives the user the impression of directly manipulating the beam model. By extending the direct manipulation paradigm to the post-processing and visualising, enhances this experience even further.

Students have successfully used the concepts introduced in ObjectiveFrame as a "Virtual Workshop" in architectural education as well as in design science education.

REFERENCES

[1]   P. Christiansson, Knowledge communication in the global network, Position paper for the July 16-20 1995 Workshop on Research Directions in Architectural Computing, Published as a chapter in a book from KLUWER in June 1996

[2]   A. Dix et al, Human-Computer Interaction, Prentice Hall International (UK), 1993

[3]   Half-life, http://sierrastudios.com/games/half-life, 2000

[4]   High Structures - A Creative Investigation (Swedish), http://www.arch.chalmers.se/projekt/high_structures/index2.html, 2000

[5]   Lindemann J., Interactive Visualisation Framework - User's guide, Report TVSM-3038, Division of Structural Mechanics, Lund University, 2000

[6]   OpenGL, http://www.opengl.org, 2000

[7]   J. Preece, Human-Computer Interaction, Addison-Wesley Publishing Company, 1994

[8]   B. Spitzak, Fast Light Toolkit FLTK, http://www.fltk.org, 2000

Part III

Appendix

Paper A.1

Initial Usability Study of ObjectiveFrame

2003

Initial usability study of ObjectiveFrame

Jonas Lindemann

September 17, 2003

Abstract

ObjectiveFrame is a 3D frame analysis tool targeted towards an educational setting. To further the use of ObjectiveFrame with a wider user group, an initial usability study has been performed. This paper describes the methods and findings of that study.

1 Introduction

A special software tool ObjectiveFrame have been developed to aid in the teaching of mechanics to engineering and design students. ObjectiveFrame is a 3D frame analysis software, with which the students can interact in real-time with the structures they create. This functionality of the software has been successfully used with prebuilt models. Problems started when the users had to create the structures themselves. The general structure of ObjectiveFrame is that of a classical finite element application, i.e. you first create element properties (Section data, material properties etc.). These properties can then be assigned to the elements in the model. To make the model flexible an element property can be assigned to many elements. The same approach is also used in conjunction with loads and boundary conditions. The advantage of the above described method is flexibility. The disadvantage is that novice users and users not familiar with finite element software have difficulties using the applications. We have seen this in the courses in which we have used ObjectiveFrame. Other issues that are problematic in ObjectiveFrame is how to interact and create the 3d frame model in an easy to use way.

The goal of this project is to investigate how the interface of ObjectiveFrame can be designed in a way that it is more accessible to a wider range of users, especially students learning mechanics at different levels. The project will not look at all aspects of the ObjectiveFrame user interface, but focus on some problem areas as described above.

2 Initial user study

The main users of ObjectiveFrame can be described as follows:

- **Architectural students** learning mechanics as a part of the courses in high structures during the final part of their education.
- **Design students** learning basic mechanics in the first terms of their education.
- **Civil engineering students** learning basic mechanics in the first and second semester
- **Teachers** in mechanics

2.1 Goals

The typical goals of the users can be described as in the following table:

2.2 Knowledge and skill level

Most of the users are quite computer literate and have little problem handling complex user interfaces. The students in architecture and design often use applications such as Adobe Photoshop, Illustrator and 3DStudio, which have very complex user interfaces. Users in civil engineering often have experience from CAD software such as AutoCAD.

User group	Goals
Architectural students	Create and study the behaviour of build-inglike structures.
Design students	Create and study the behaviour of structures.
Civil engineering students	Create and study the mechanical behaviour of beams and beam structures. Create beam topology for input to other finite element software.
Teachers	Create illustrative models for use in the teaching of mechanics.

Table 1: Typical user goals

1. Create the given structure. Add additional members to the structure so that the deflection of the top member is maximum 10 mm.
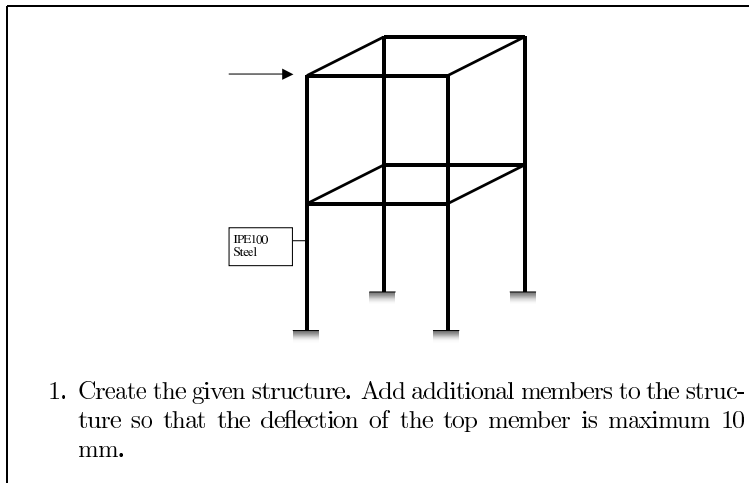
Figure 1: A typical task to be solved using the ObjectiveFrame application

2.3 Work environment

The student work environment can be divided into two groups. Civil engineering students often work in a lecture and exercise based form, attending lectures given by staff at the departments and then taking part in scheduled 2 hour exercises. The students in architecture and design often work in larger projects attending lectures and then applying the knowledge in the projects. Scheduled exercises exists in some courses.

ObjectiveFrame will be used to enhance the understanding of certain concepts in mechanics, when the students are doing project work in these courses.

2.4 Objects and tasks

The assignments given in the courses are typical to study some kind of structure and then discuss some phenomena. A typical task is shown in figure 1.
The objects involved in this exercise are:

- Structural nodes (connection points)
- Beam elements

- Forces
- Boundary conditions or constraints
- Simulation
- Results
- View

Typical tasks in the exercise are:

- Modifying the viewpoint
- Creating, modifying and deleting nodes
- Connecting nodes with beam elements
- Removing beam elements
- Modifying beam element properties
- Adding, modifying and deleting loads
- Adding, modifying and deleting boundary conditions
- Running simulation
- Viewing and interpreting results
- Modifying model after interpreting results

3   Usability evaluation

To understand the behaviour of structures subjected to loads the 3d beam analysis program ObjectiveFrame has been developed. One of the features of this program is that it can visualise the response of a structure to a user-defined load in real-time. This makes it suitable as an educational tool in design science, architecture and structural mechanics.

When ObjectiveFrame has been used in courses for architects and design students, it was apparent that usability problem existed. This section describe the current user interface and the issues related to its design.

3.1   General user interface description

The main user interface in ObjectiveFrame is the workspace, see figure 2 The workspace is the "workbench" on which the beam models are assembled and consists of an axis and a transparent grid. The user can change the workspace size to fit the model constructed.

The main user interface is centered around the 3d view showing the "virtual" workbench. Around the 3d view the main toolbars of ObjectiveFrame are placed: the edit toolbar for manipulating existing objects in the view, the model toolbar for creating new objects in the scene and the view toolbar for manipulating the current 3d view.

Object placement, movement and creation are done using a special 3d cursor. The position of the 3d cursor is determined by mapping the mouse coordinates on to the 3d grid.

3.2   Cursor placement

A central concept in ObjectiveFrame is the nodal point, where beams are connected and loads and boundary conditions are applied. The nodal point has to be placed in 3d in some way. To implement node placement ObjectiveFrame uses a special 3d cursor, which can be moved using the mouse. Using a mouse to place 3d objects has certain drawbacks, because it only supports 2 degrees of freedom, x and y. The current version of ObjectiveFrame projects a ray from the pointer position on the screen to the plane of the "virtual workbench". This enables cursor movement in the plane. To support movement out of the plane the cursor must be locked in the xy-plane. Cursor locking is done by holding down the [Ctrl] key.

The problem with node creation and cursor placements was that the information of locking was not visible in the user interface. This made it harder for new users to begin using the application. Neither he process of creating nodes is clear from looking at the user interface. The cursor did not provide any clues of what to do when the node creation mode in ObjectiveFrame was selected, which lead to many questions from the students.

Another problem with the cursor placement method was the mapping of mouse movement when placing a node out of plane (z-direction). It is not always possible to figure out the mapping of the mouse movement to the movement of the 3d cursor. Figure 3 illustrates these problems.
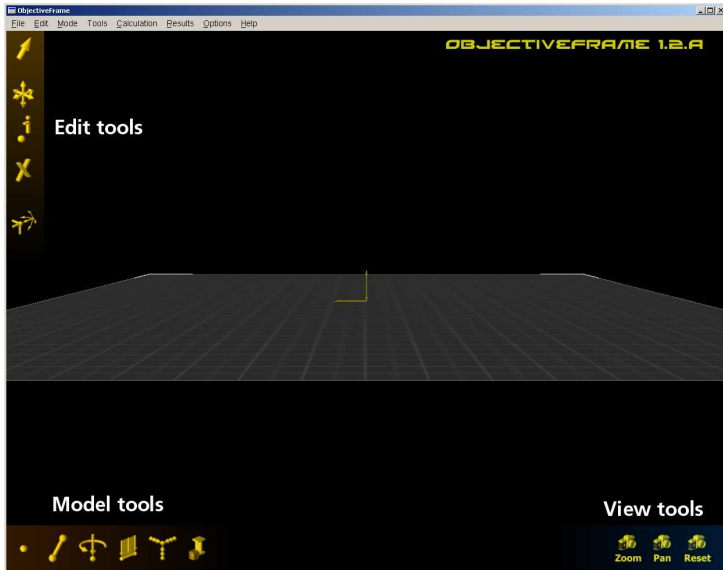
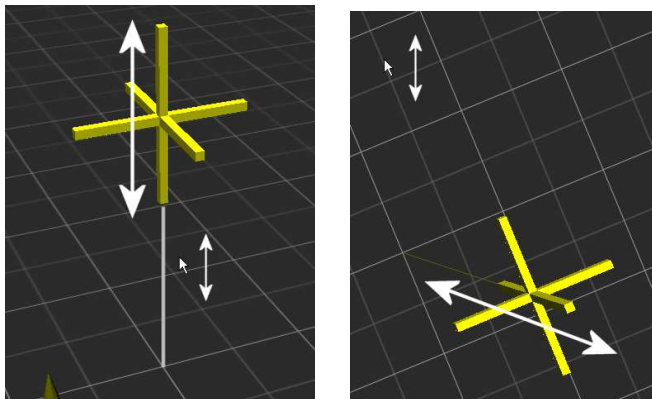Figure 2: Current ObjectiveFrame user interface

Figure 3: The left figure shows a good mouse pointer mapping of the 3d cursor. The right figure shows a problematic mapping when the viewing angle changed.

3.3   Object selection

Modifying and removing objects in ObjectiveFrame requires objects to be selected. Selection is handled using the select tool in the edit toolbar. When the mouse pointer is over a selectable object, the object is highlighted in white. Clicking an object changes its colour to yellow. The highlight effect is providing good feedback for the users, indicating what objects are selectable. The method for indicating that an object is selected causes a lot of problems. The main problem is the conflict with the colouring scheme in ObjectiveFrame; in ObjectiveFrame it is possible to assign different beams with colours indicating specific element properties. Assigning yellow for a given element property will greatly confuse object selection, see figure 4
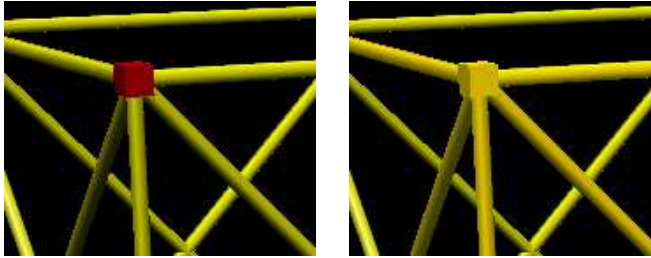
A-6

Figure 4: Object selection problems

3.4 Moving nodes

The current method of moving nodes is as follows:

1. Select the the points to be moved using the select tool.
2. Select the move tool.
3. Place the 3d cursor at the initial movement point.
4. Drag the 3d cursor for moving the points. Use [Ctrl] for moving the points in the z-direction.

The method does not provide any clues of how it should be used and many users click on the movement tool and try to drag the points directly, which fail, or does not provide the desired outcome.

3.5 Applying loads and boundary conditions

The main usability problem with ObjectiveFrame is due to the fact that is was initially not designed to be used in the teaching of design and architectural students, but an experimental user interface for a 3d beam analysis code. The target users were engineers or engineering students. These user groups often require the applications to be very general and flexible, which often lead to more complex user interfaces.

Loads and boundary conditions in ObjectiveFrame are handled using special windows (see figure 5), where the users can create load cases and apply these to selected nodes or elements.
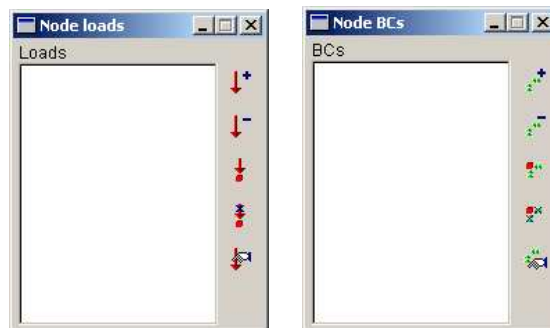

Figure 5: Load and boundary condition windows

Architecture and design students are often unfamiliar with the concepts of load cases. The concept of applying a load defined in the window to many nodes at the same time is also confusing. A more direct approach would be preferable.

4   Revised user interface

To enhance the current user interface a new prototype interface for ObjectiveFrame has been designed and is described in this section. The prototype will focus on the following areas as described in section 3:

- Node creation and cursor placement
- Object selection
- Moving and copying nodes
- Applying loads and boundary conditions

4.1   Conceptual design

The conceptual design in ObjectiveFrame is centred around the virtual shop concept. In this conceptual model, sthe user assembles a model on a workbench. As an aid in assembling a model, the user also has a number of building blocks available. ObjectiveFrame uses a ball-and-stick model for creating three-dimensional beam models. This model consists of two main constructional elements: the ball and the stick. The ball is used as a connection point to which the sticks are connected to form a structure. The ball and stick model can be found in chemistry for assembling molecular structures and in engineering for assembling space frames, see figure 6
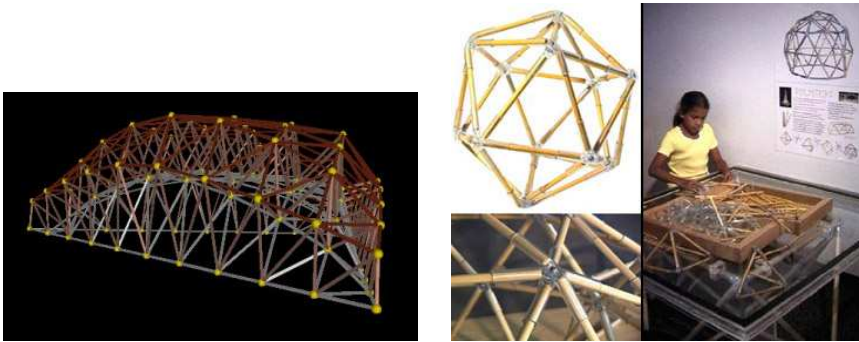

Figure 6: Ball-and-stick models

Table 2 describes the main building blocks in ObjectiveFrame.
These building blocks present the user with a construction kit for beam and truss structures. It can be compared to having a Meccano kit, see figure 7 with simple elements that can be assembled into structures that are more complex.

4.2   Interaction design

The ObjectiveFrame prototype was created using a combination of tools and libraries. The 2d user interface was created using the FLTK [2] library and the user interface designer FLUID, that comes with this library. The 3d user interface was implemented using the Ivf++ [3] visualisation library. These tools enable quick and easy creation of a functional prototype. It is also easy to change functionality of the prototype.
The main window of the prototype is divided into three parts: 3D view, toolbar, and status view, as shown in figure 8. In the toolbars, functions for setting the current state and other functions are placed. The current state is indicated using a checkbox in the button. In the status view, the users are given short information on the selected state or function. There is also a coordinate display showing the current coordinate of the 3D cursor. Hints are also used to display additional information about the state and functions. The appearance of the toolbar and status views will probably be changed in the final version.
The user interface of the ObjectiveFrame prototype is designed as a direct manipulation interface, i.e.

Representation	Name	Description
	Node	The node is the "ball" in the conceptual model of ObjectiveFrame. Beams, loads and boundary conditions can be attached to the nodes.
	Beam/bar	The beam is the "stick" in the conceptual model. It is also the main structural element in the finite element model.
	Load/Force	The load represents an external force applied to the structure and is represented by an arrow pointing in the direction of the force.
	BC	A boundary condition (BC) representation a constraint to the structure. For example a node fixed support.

Table 2: ObjectiveFrame building blocks
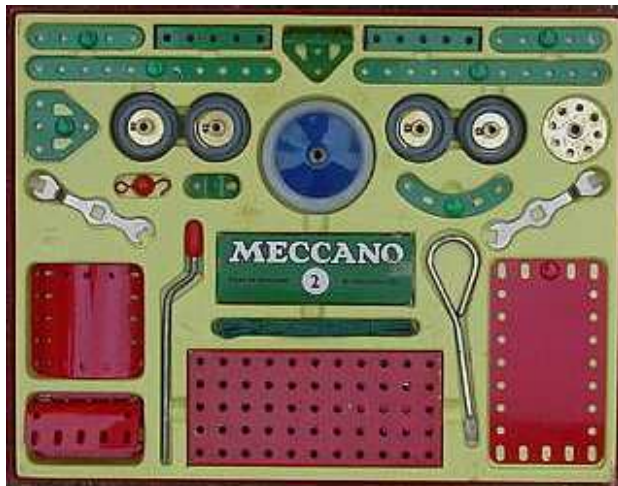
Figure 7: A Meccano kit. Image from [1]

- Continuous visual representation of objects and current operations.
- Physical actions instead of complex commands.
- Fast, incremental and reversible actions.
- Easy for beginners to start using the system.

In the ObjectiveFrame prototype direct manipulation is used in most operations, for example when moving a node will show a node representation being moved on the screen. Rotating the view is updated directly when the user moves the mouse.

Node creation and cursor placement

To make the node creation more intuitive the 3D cursor has been enhanced. The cursor now shows the object to be placed in the middle and has arrows showing the direction of allowable movement, see figure 9

Figure 8: Prototype main window

Figure 9: Updated 3d cursor

The first step when creating a node is placing the node in the xy-plane. The cursor shows four arrows representing the allowable movement directions in the xy-plane. When the user clicks at a position in the xy-plane the 3D cursor will after that lock 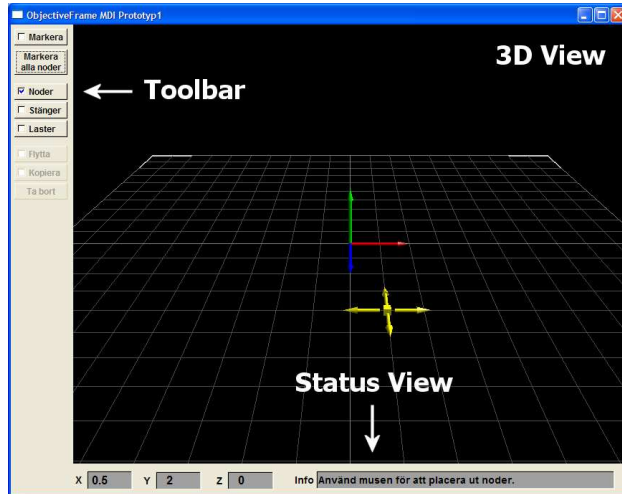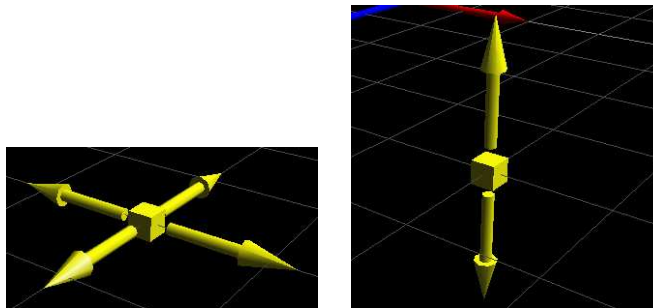at the selected position and will show two arrows in the z-direction indicating that the allowable movement now is in the z-direction. Clicking a second time will create a node at the specified position. The cursor will then return to the initial state, locked in xy-plane to allow the user to create additional nodes. A state diagram of the process is shown in figure 10.

In the current version of ObjectiveFrame, cursor movement in the z-direction was done by taking mouse y movement and scaling it against the workspace size. This method produced mapping problems when used in certain view angles as described earlier. To solve this, the intersection point from the screen (x, y) position to a special plane containing the xy-position selected in the first click is calculated. This method will provide a more correct mapping of the mouse movements against the 3D cursor movement. The method is illustrated in figure 11.

Object selection

Most functions such as moving, copying and deletion in ObjectiveFrame operate on an object selection. Object selection in the prototype application is done by clicking on a selectable object. A Selectable object is highlighted when the mouse pointer is moved over it in the workspace.
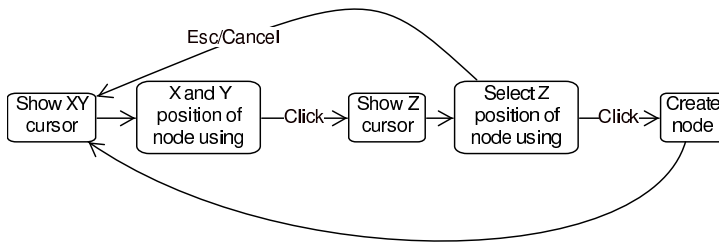
Figure 10: Node creation state diagram
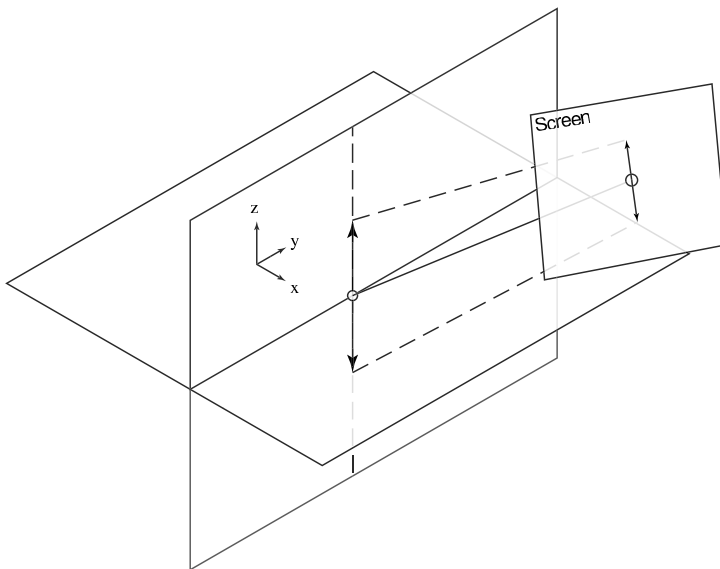
Figure 11: Z-mapping method

Clicking on additional objects will add these to the selection. If a user clicks on an object which is already selected the object will be removed from the selection.

Moving and copying objects

Moving and copying operates on selected nodes. To move or copy a selection of objects, the user clicks on it, after which the same method as in node creation is used to move it. To give the user a better understanding of what to do, a special cursor shape is shown over a node when the mouse pointer is over it, see figure 12. The selection highlight is moved along with the 3d cursor when the user has clicked on the first node, see figure 13

Creating loads

In the current version of ObjectiveFrame, loads are created in the load windows. The direction of the load is specified in x, y and z directions. Setting a negative value can also affect the direction

A-11
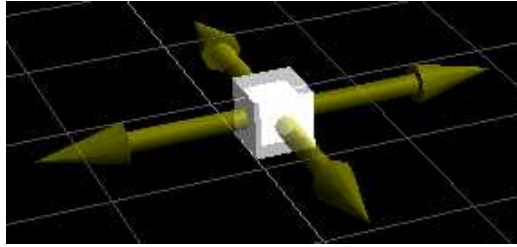
Figure 12: Move cursor

Figure 13: Moving/copying objects

of the load. To make the load creation more direct a special method for visually applying a load has been developed.

The difficulty of creating a load in 3D is to be sure about its direction, due to the fact that the perspective transform distorts angles and distances, making it hard to see the direction of a vector. The prototype implements a special widget aiding the user in directing the load visually in 3d.

The load creation process starts by moving the mouse pointer over a node. When the mouse pointer is over a node a special load cursor is shown i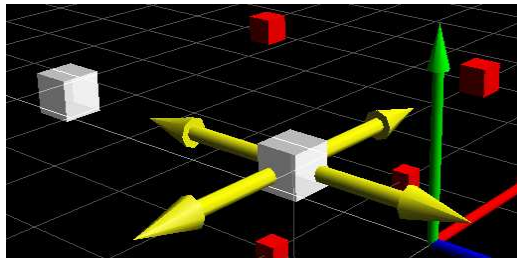ndicating that a load can be applied to the node. Clicking on the node will show the "widget" for directing the load. The "widget" consists of one vertical circle and a horisontal circle. On the vertical circle, the load is attached. The load can glide on the vertical circle providing the first angle of the load. The vertical circle is attached to the horisontal circle, which is rotated in the second click, providing the second angle for load placement. The complete process of applying a load is shown in figure 14.

4.3   Graphical design

Many of the guidelines available for graphical design apply only to 2D user interfaces. Some guidelines, such as those for colour and icons can equally well be applied to 3D user interfaces. The focus in this evaluation has been on interaction design in the 3D user interface. Due to the lack of guidlines some new solutions have been proposed.

Selection and highlight

A lot of work developing this prototype was done developing a new way of visualising the selection process. The previous method of using a single colour was problematic, as the models created often use colour to indicate material properties. To solve this, a special method of object selection has been developed. In this method, an object selected is rendered twice. First the object is rendered using its original colours, secondly the object is rendered scaled to 120% and using a light grey colour. The result is then composited together, yielding an object which is highlighted regardless of initial object colour. Figure 15 illustrates the new selection method.
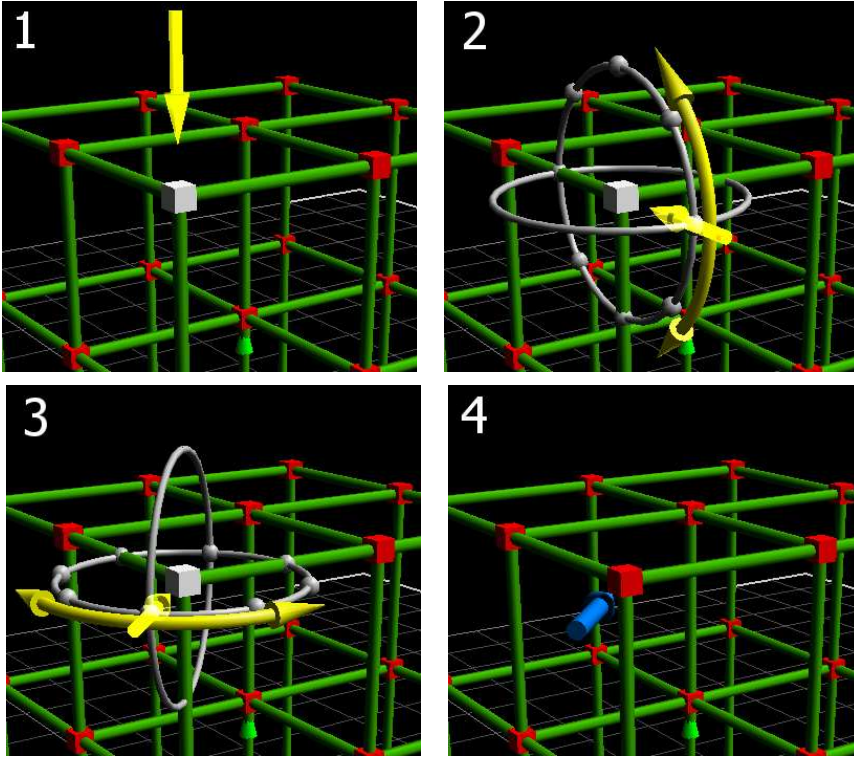
Figure 14: Applying loads
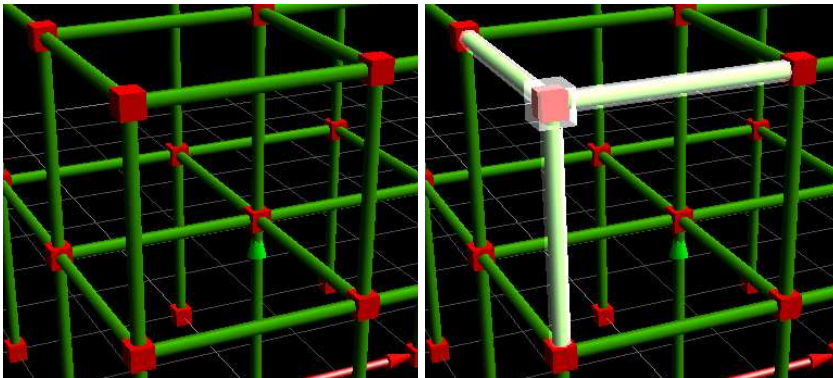
Figure 15: Object selection using a two pass rendering technique.

Colour and lighting scheme

The building blocks in the prototype are given distinct colours or materials to make it easy for the user to distinguish between them. Nodes use a red material, beams, a green material, and loads a blue material. Cursors and 3D icons all use a yellow material. Helper objects such as the guide circles in the load application method use a dull grey appearance, so that they don't draw the focus

away from the cursor icons. The workbench also uses darker grey colours, so that the grid won't clutter the created model.

The entire model in the prototype is rendered with lighting. This is to enhance user experience of "assmebling" a real model. The light is attached as a headlight to the current view direction. This will always produce a well-lighted model without any dark areas.

5   Usability study

The usability study is divided in two parts. The first part is a normal usability study involving real users evaluating the prototype. This method is not allways possible to do due to lack of resources or time. The second part describes a heuristic evaluation using Shneiderman's 8 golden rules for interaction design [4]. In this method the user interface is evaluated using a set of rules to determine the usability of the interface. Even if this method does not involve any users it can point out many problems in a user interface design.

5.1   User test

The user test was conducted using the prototype described in chapter 4. Four civil engineering students were test users. One of the test users were already experienced with the current version of ObjectiveFrame and the other three users had never used ObjectiveFrame before.

Test setup

The test setup used a standard PC with keyboard, mouse and 3D graphics support. A standard DV video camera was used to record the screen, keyboard and mouse. To accurately capture user input a screen capture program recorded the user interactions for each task. The CamStudio [5] software was used, which enables loss-less screen recording without disturbing the user.

Test procedure

Before starting the test, the users were informed of the objective of the test. They were also informed that questions on the prototype were not going to be answered during the test, unless the test person wascompletely stuck.

The users were given seven tasks to complete. Tasks 1 - 4 covered node creation, node selection, copying and moving nodes. The tasks used a simple structure as shown in the left image of figure 16. In tasks 6 and 7 a predefined beam structure, as shown in the right image of figure 16 is used to evaluate beam selection and load creation.

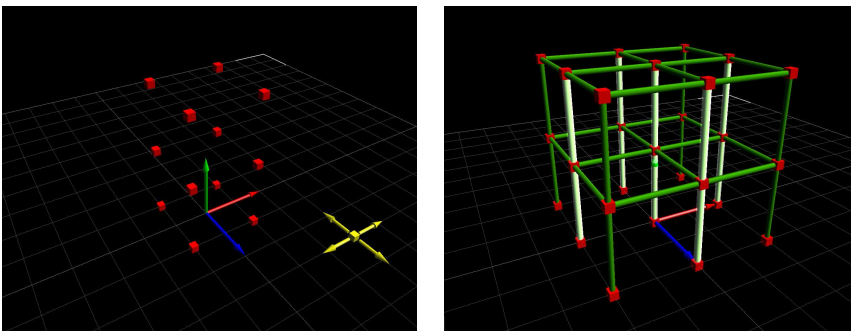Figure 16: Structures used in task 1 - 4 (left), 6 and 7 (right)

For each task a description on how to manipulate the view were given. The user interface for view manipulation has not been determined yet. In addition, the users were told that all operations in the prototype could be cancelled using the [Esc] key. The users were also instructed not to turn to the next page before the current task was completed.

The user tasks were evaluated by studying the recorded screens and video. During the recording notes were taken about important events. Timing of specific tasks were done by studying the screen capture video and determining three timings:

Inactivity Amount of time where the user does not move the mouse. During this time the user is probably studying the task description.

Pre-operations Amount of time before the user selects the studied operation. For example moving around changing the view.

Operational time Amount of time for completing the task.

These timings can be found in the diagrams accompanying each task in the following sections.

Task 1 - Creating nodes

In task 1, the user should create a set of nodes according as shown in the left image of figure 16. The two-step node creation procedure seems to work. There is an initial time spent figuring out how to move the mouse, but when the first node has been created, the others are created very quickly. One of the users placed the first layer of nodes correctly but had problems placing the next layer correctly. The Esc-key was used successfully to abort the node creation process. An interesting side note: Instead of creating the nodes one by one, one user used the copy function for copying an entire node layer.
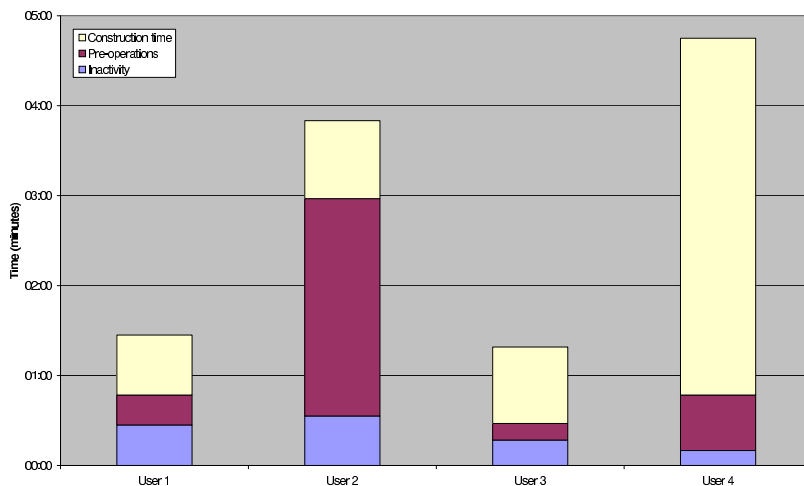
Figure 17: User times for task 1 - creating nodes

Most of the problems with the prototype in task 1 can probably be solved by enhancing the information displayed in the status view. Some kind of state indicator showing the user where in the node creation process he or she is would be desirable.

Task 2 - Selecting and deleting nodes

In task 2, the user should select and delete the top layer of nodes in the given structure. The selection method seems to work fine. Some of the users used the view commands in this exercise to modify the view. One of the users didn't use the multiple selections, but instead deleted the nodes one by one.

The selection process can be:

- Clicking on objects will add them to the selection. Clicking on already selected objects will remove them from the selection. Clicking outside the selection will remove all objects from the selection.
- Clicking objects will select them. Clicking on a non-selected object will deselect the previous one. Multiple selections are handled using modifier keys such as [Shift] and [Ctrl]. This method is common in many Microsoft applications, such as Word or Excel.

The first method mentioned was used in the prototype. Users do not seem to have any problems with it. The highlighting of object when the mouse is over them also seem to accelerate the selection process.
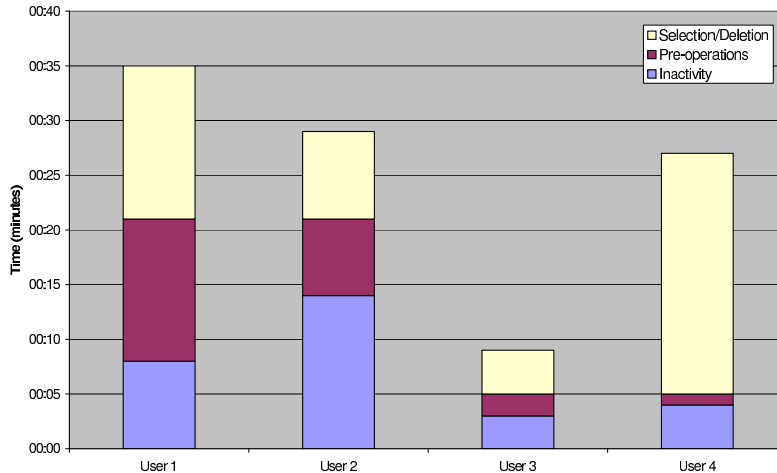
Figure 18: User times for task 2 - Selecting and deleting nodes

Task 3 - Copying nodes

In this task, the users should copy the top four nodes in the given structure. The copy method works in approximately the same way as with moving nodes. Users seem to have little problem with this method. The special cursor that is shown when the mouse is over a node works well. Some of the users also used the select-deselect functionality of the prototype.

Most of the problems with the prototype in task 3 can probably be solved by enhancing the information displayed in the status view. Some kind of state indicator showing the user where in the copy process he or she is, would be desirable. The selection process should perhaps also be reviewed to solve some of the problems in this task.

Task 4 - Moving nodes

In this task, the users should move two nodes from the left and right sides. The move method works in approximately the same way as with moving nodes. Some users were probably expecting the selection to be removed when the move operation was finished. There were also some attempts to deselect nodes by clicking on the select button after the operation was finished. The special cursor that is shown when the mouse is over a node works well. Some of the users also used the select-deselect functionality of the prototype.

Most of the problems with the prototype in task 4 can probably be solved by enhancing the information displayed in the status view. Some kind of state indicator showing the user where in the node creation process he or she is, would be desirable. The selection process should perhaps also be reviewed to solve some of the problems in this task.
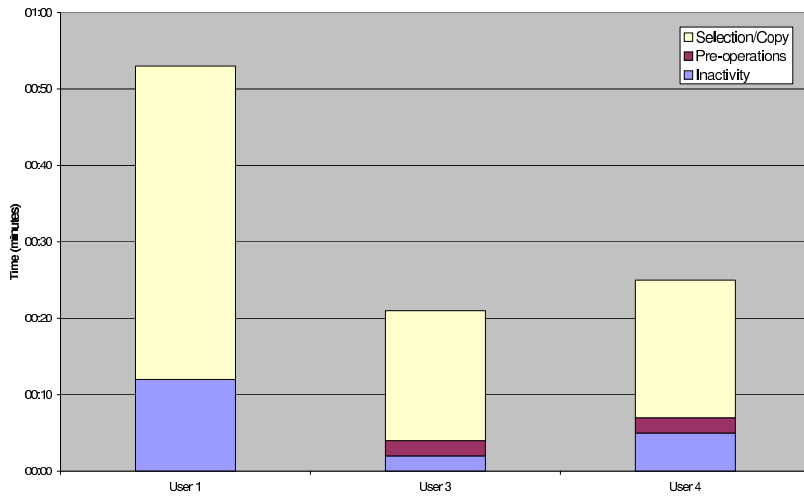
Figure 19: User times for task 3 - Copying nodes
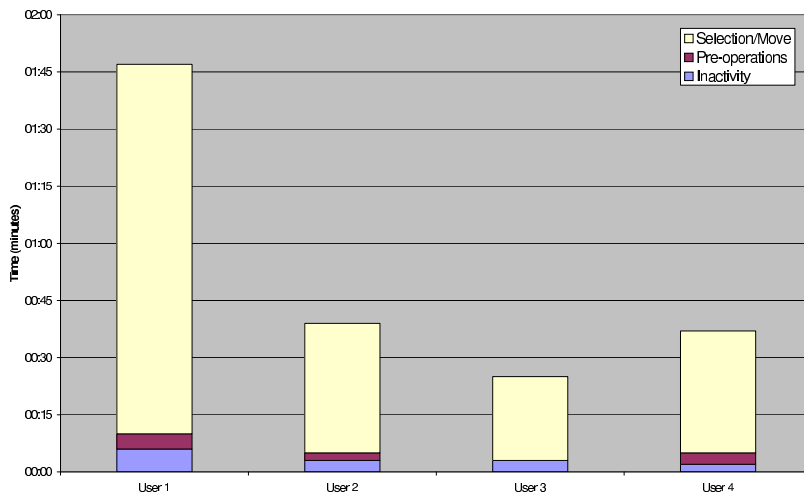
Figure 20: User times for task 4 - Moving nodes

Task 5 - Creating beams

In this task, the users should create beams according the given structure. This process seems to have no problems. Highlighting of allowable objects seems to aid in the process of selecting the nodes for the beams. One user created beams in the wrong place, and used the select tool and deleted these without problems.

A-17
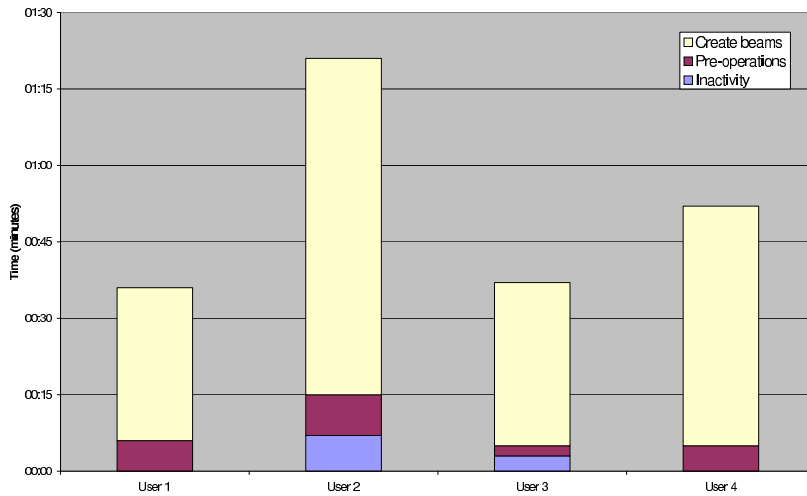
Figure 21: User times for task 5 - Creating beams

Task 6 - Selecting and deleting beams

In this task, the beams shown in the picture should be deleted. Users quickly selected the beams. If the beams are hidden in the 3D view, they change the view. Highlighting of selectable objects is speeding up the selection process. One user tried to delete the axis indicator. In addition, some problems with using the view functions were noted.

Visible buttons for changing the view should be added to aid users not familiar with how to change the view using the mouse and modifier keys. The axis indicator should probably also be made smaller or more different from the beams.
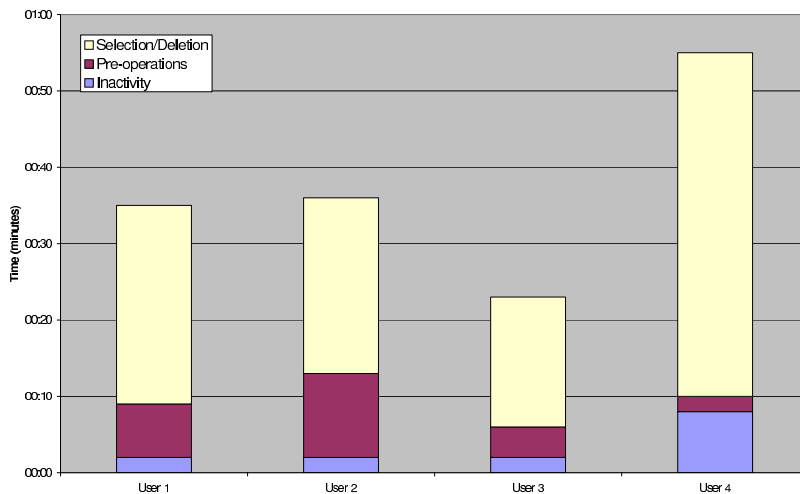
Figure 22: User times for task 6 - Selecting and deleting beams

Task 7 - Creating loads

In this task, the users should add loads to the structure given. All of the users seem to figure out how to use the load widget. There seems to be a mapping problem in the widget rotational control. Two of the users had difficulties figuring out how to rotate the load. There were also problems when a load was created in the wrong direction, but this was mainly related to the fact that the prototype does not support deleting nodes.

The mapping for the rotational widget should be reviewed, so that they are more easily interpreted.
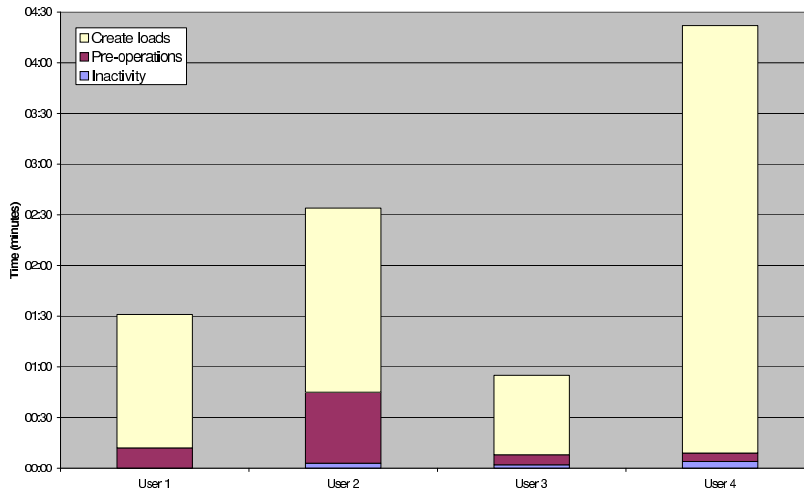
Figure 23: User times for task 7 - Creating loads

5.2 Heuristic evaluation

In this section the prototype application is evaluated using the eight golden rules for interaction design according to Shneiderman [4].

Consistency

The prototype takes advantage of consistency by using the same methods for moving and placing objects in 3D. Node creation, move and copy functions all use the same cursors and methods. Cursors are also shown using the same colour and graphical design. The Esc-key can be used throughout the different operations in the prototype.

Shortcuts for experienced users

The prototype supports experienced users by providing shortcuts to the commands. By pressing special keys, the experienced user quickly can switch between the different states of the application. In addition, movement in the Z-direction can be reached by holding down the [Shift] key.

Informative feedback

Feedback is provided in the status view of the application. In some cases, the feedback could be improved. For example, when placing objects in 3D there should be some indicator of the state of the process. In addition, confirmation dialogues should perhaps be added before objects are deleted. This could be implemented as an option in the application, to let more experienced users delete without confirmation.

Design dialogues to yield closure

The prototype application does not contain dialogues, but the process of creating nodes, copying and moving objects can be seen as a kind of dialogs. As such, there should be more feedback of the process, as discussed in the previous section.

Offer error prevention and simple error handling

Many errors in the prototype are prevented by only allowing certain actions to take place, such as only allowing nodes to be selected when creating beams. The highlighting scheme used in the object selection process is also helping the user to place the mouse more accurately.

Permit easy reversal of actions

The prototype application only supports very limited action reversal. This reversal is in the form of the Esc-key permitting the current operation to be cancelled. This is an area which should be enhanced in the new version of ObjectiveFrame.

Set the user in control

The prototype is designed in such a way that the user always has control. The user is given a set of building blocks that can be used to create any desired structure. The purpose of the application prototype is to aid in this creation process, not interfere. A minimum of dialogues and windows are used, so that structure creation can be compared to assembling a physical structure.

Reduce short term memory use

Most operations and commands are clearly visible. Most information is available in the user interface. Hint boxes provide additional information on the functionality of the buttons and tools. The user is provided with additional instructions in the status view about what he has to do to complete the operation.

6 Conclusions

ObjectiveFrame has shown great potential as an educational software. It was initially designed to be a general application tool with engineers as the main user group. Use in courses for architectural and design students has shown that the concepts used in ObjectiveFrame was difficult to use for these students.

This work has illustrated that ObjectiveFrame can easily be modified to be a more direct application, more suitable for an educational setting. It has also been shown that by doing a simple user testing and a heuristic analysis a lot of information on the problems of a user interface can be found.

Through the information provided in the user test and the heuristic evaluation, the following points will be improved in the future versions of ObjectiveFrame.

State information Some kind of indicator should be added to display what state the current operation is in. In the node creation process, information on which step the application is currently in should be clearly visible together with information on how to complete the step.

View handling The view handling should be improved by supporting view manipulation using visible buttons in the application, in addition to using the mouse and modifier keys.

Improved axis indicator A more distinguishable axis indicator should be designed so that it would not be mistakenly taken for a beam object.

Selection An improved selection visualisation is implemented, so that is more clear what objects are selected in the model.

Confirmation dialogues Critical operations should be preceded by optional confirmation dialogues.

Mapping of rotations The mapping of rotation angles when creating loads should be improved, to make the load placing a more intuitive process.

References

[1] The MECCANO Light Red and Green Period,
http://www.meccanonut.com/lightred/index.htm, 2003

[2] The Fast Light Toolkit Home Page, http://www.fltk.org, 2003

[3] The Interactive Visualisation Framework - Ivf++,
http://www.gorkon.byggmek.lth.se/ivfweb, 2003

[4] Shneiderman B., Designing the User Interface – Strategies for Effective Human-Computer
Interaction, Addison-Wesley, 1998

[5] Rendersoft CamStudio 2.0, http://www.rendersoftware.com/products/camstudio, 2003

Paper A.2

Interactive Visualisation Framework − Ivf++

http://www.gorkon.byggmek.lth.se/ivfweb, 2003

Ivf++ – An Extendable OpenGL Visualisation Framework

Jonas Lindemann

September 17, 2003

Abstract

Ivf++ is a C++ application framework for developing OpenGL applications. The library focuses on extensibility and ease of use. The library also enables direct use of OpenGL function calls. The library consists of approximately 240 classes and is released under the LGPL license.

1 Introduction

Visualisation applications often use the OpenGL graphics library [9] for rendering real-time 3D graphics. OpenGL is a software interface to graphics hardware consisting of 150 distinct commands. To be platform indpendent OpenGL does not have commands for creating windows or providing user input. To use the OpenGL redering commands, a window with a suitable rendering context must exist. Creating windows and rendering contexts is different depending on which platform is used, making it hard to implement platform indpendent visualisation applications. Drawing in OpenGL can also be complex. OpenGL can be seen as the assembly language of 3d graphics, requiring knowledge of transformation matrices and linear algebra, to fully utilise the library. Ivf++ or Interactive Visualisation Framework [5] was developed to make it easier to use OpenGL and in the same time enable the user to fully utilise the advantages of the library. The library is also designed to be modular, to be seamlessly integrated with other libraries. The different libraries included in Ivf++ are described in table 1 and in figure 1.

library	Description
ivfmath	3D math classes
ivf	Scenegraph library
ivfui	User interface library
ivffile	3D file loaders
ivfimage	2D image file loaders
ivfctl	Support for animation controllers
ivf3dui	Support for 3d user interface controls
ivfwidget	User interface abstraction library
ivffltk	FLTK integration library
ivfwin32	Win32 integration library
ivfext	Ivf++ extension library

Table 1: Ivf++ libraries

2 Creating a basic OpenGL application

To create an standard OpenGL application using Ivf++, the `ivfui`-library can be used. This library encapsulates all the windowing details and event handling needed for a basic
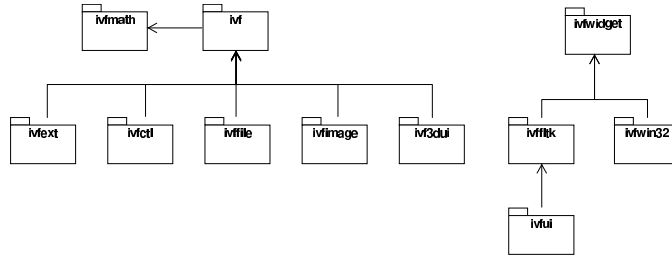
Figure 1: Ivf++ library dependencies

OpenGL application. Implementing a simple OpenGL application using the `ivfui`-library usually consists of four steps:

- Adding the `ivfui` include directives.
- Declaring a `CIvfWindow` derived window class.
- Declaring and assigning events to be used.
- Implementing the main procedure.

The necessary include directives for a `ivfui` based application are:

```
#include <ivfui/IvfApplication.h>
#include <ivfui/IvfWindow.h>
```

Next, we derive a window class from the `CIvfWindow` class. To handle events, three event classes are added to the inheritance list and the virtual methods from these classes are added to the window class definition. Events in Ivf++ are C++ class instances containing a single event method.

```
class CExampleWindow: public CIvfWindow,
    CIvfInitEvent,
    CIvfResizeEvent,
    CIvfRenderEvent {
private:

public:
    CExampleWindow(int X, int Y, int W, int H);

    virtual void onInit(int width, int height);
    virtual void onResize(int width, int height);
    virtual void onRender();
};
```

To receive events for the defined virtual methods, they have to be registered with the base class. This is done in the class constructor.

```
CExampleWindow::CExampleWindow(int X, int Y, int W, int H
    )
    :CIvfWindow(X, Y, W, H)
{
    addInitEvent(this);
    addResizeEvent(this);
    addRenderEvent(this);
}
```

It is not allowed to call any OpenGL commands in the class constructor of an `ivfui`-based application. A valid rendering context does not exist when the constructor is called. To handle initialisation of objects and OpenGL state information the `onInit` is used instead. This method is called after the window has been created and a valid rendering context exists. The `onInit` event method has 2 parameters `width` and `height`, representing the initial size of the window. In this example the default Ivf++ OpenGL settings (depth buffer and lighting enabled) is used and no other initialisation is needed, so an empty `onInit` is added (Will be used later on).

```
void CExampleWindow:: onInit(int width, int height)
{

}
```

Window resizing is handled in the `onResize` event method. The event method takes 2 parameters, `width` and `height` for the new size of the window. In this example, the `onResize` method is used to setup the viewport and view transform using OpenGL commands.

```
void CExampleWindow:: onResize(int width, int height)
{
    glViewport(0,0,width,height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}
```

To display something in the window the example window must implement the `onRender` event method. From this method the actual OpenGL rendering will be done. It should be noted that Ivf++ will maintain the matrix state between OpenGL calls by calling `glPushMatr:` and `glPopMatrix` before and after the `onRender` method. Clearing of the screen is also automatically handled by Ivf++ (This behaviour can be overridden by the `onClear` event method). In this example a simple coloured quadrilateral is rendered.

```
void CExampleWindow:: onRender()
{
    glDisable(GL_LIGHTING);
    glBegin(GL_POLYGON);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f (0.25, 0.25, 0.0);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f (0.75, 0.25, 0.0);
    glColor3f(1.0f, 1.0f, 0.0f);
    glVertex3f (0.75, 0.75, 0.0);
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f (0.25, 0.75, 0.0);
    glEnd();
}
```

To make it easier to handle allocation and deallocation of objects, Ivf++ implements a smart pointer system. A smart pointer is a template based class that mimics a normal pointer declaration, adding function for automatically initialising and destroying the class instance. The Ivf++ smart pointer also handles the reference counting scheme implemented in Ivf++. Smart pointers are created with the `IvfSmartPointer` macro. In this example a smart pointer is declared for the example window by adding the `IvfSmartPointer` macro just before the class declaration. This defines a smart pointer class `CExampleWindowPtr`, which is used later on when instantiating the window.

```
IvfSmartPointer(CExampleWindow);

class CExampleWindow: public CIvfWindow , ...
```

The implementation of the window class is now complete. To finish the example a main routine has to be implemented. First, an instance of an application object is created. The application object encapsulates the main event loop of the application. It also sets the desired default rendering context for the created windows; in this example a double buffered colour display supporting RGB colour.

```
CIvfApplicationPtr app = new CIvfApplication(IVF_DOUBLE|
    IVF_RGB);
```

In the next step the window class defined earlier is instantiated and displayed using the show method.

```
CExampleWindowPtr window = new CExampleWindow
    (0, 0, 512, 512);
window->setWindowTitle("Ivf++ OpenGL application");
window->show();
```

The application loop is entered by calling the run of the application object app. When all application windows are closed, the run methods will return.

```
app->run();
```

Compiling and executing the application at this point will show black window with a single coloured quadrilateral. In every event method OpenGL calls are guaranteed to have an active rendering context. The complete example with sample OpenGL rendering code is shown below:

Listing 1: Ivf++ Standard OpenGL application

```
#include <ivfui/IvfApplication.h>
#include <ivfui/IvfWindow.h>

IvfSmartPointer(CExampleWindow);

class CExampleWindow: public CIvfWindow ,
    CIvfInitEvent ,
    CIvfResizeEvent ,
    CIvfRenderEvent {
private:

public:
    CExampleWindow(int X, int Y, int W, int H);

    virtual void onInit(int width , int height);
    virtual void onResize(int width , int height);
    virtual void onRender();
};

CExampleWindow::CExampleWindow(int X, int Y, int W, int H)
    :CIvfWindow(X, Y, W, H)
{
    addInitEvent(this);
    addResizeEvent(this);
    addRenderEvent(this);
}

void CExampleWindow::onInit(int width , int height)
{

}

void CExampleWindow::onResize(int width , int height)
```

```
{
    glViewport(0,0,width,height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}

void CExampleWindow::onRender()
{
    glDisable(GL_LIGHTING);
    glBegin(GL_POLYGON);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f (0.25, 0.25, 0.0);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f (0.75, 0.25, 0.0);
    glColor3f(1.0f, 1.0f, 0.0f);
    glVertex3f (0.75, 0.75, 0.0);
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f (0.25, 0.75, 0.0);
    glEnd();
}

int main(int argc, char **argv)
{
    CIvfApplicationPtr app = new CIvfApplication(IVF_DOUBLE|IVF_RGB);

    CExampleWindowPtr window = new CExampleWindow(0, 0, 512, 512);
    window->setWindowTitle("Ivf++_OpenGL_application");
    window->show();

    app->run();

    return 0;
}
```

Listing 1: Ivf++ Standard OpenGL application

3   The Ivf++ Scene graph

To make it easier to use OpenGL, the core library of Ivf++ implements a scene graph. A
scene graph is a structure containing nodes in a tree-like structure. To render a scene graph
the tree is traversed from top to bottom. There are two kinds of objects in an Ivf++ scene
graph, primitive objects and composite objects. Primitive objects can render themselves
without dependencies on other objects, for example sphere, cube, cylinder and triangle set
objects. Composite objects are objects that contain other objects, making the contained
objects inherit the state of the composite object. Examples of these objects are aggregate
objects and transform objects. Appearance, view transforms and state information are not
included as objects in the scene graph, but as attributes on the objects in the scene graph or
as separate objects. The main reason for this is to keep the scene graph as simple and easy
to understand as possible. Figure 2 illustrates the relationships between scene graph objects
and other objects in Ivf++.

The simplest way to use the scene graph nodes in Ivf++ is to use a CIvfComposite class
as the root node for the scene graph. In addition to this class we need some additional Ivf++
classes. The following include directives are added:

```
#include <ivf/IvfComposite.h>
#include <ivf/IvfCamera.h>
#include <ivf/IvfMaterial.h>
#include <ivf/IvfCube.h>
#include <ivf/IvfLighting.h>
#include <ivf/IvfLight.h>
```

In the class definition we add member variables for the scene graph root node m_scene,
perspective viewing m_camera and lighting m_light.
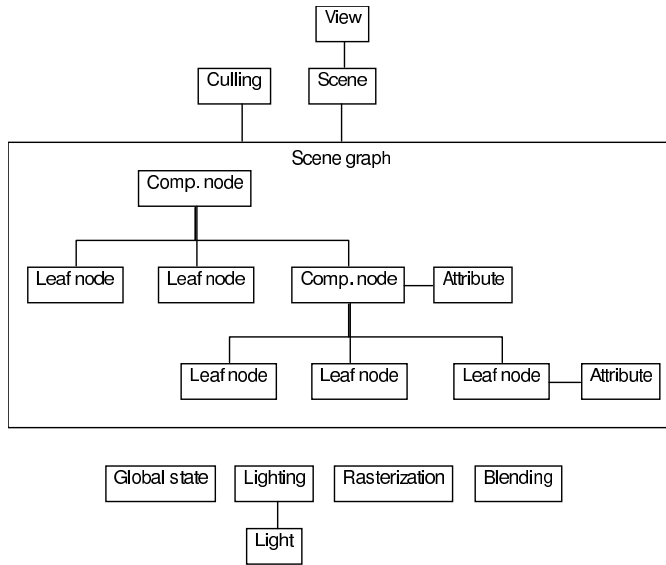
Figure 2: Ivf++ scene graph relationships

```
class CExampleWindow: public CIvfWindow,
    CIvfInitEvent,
    CIvfResizeEvent,
    CIvfRenderEvent {
private:
    CIvfCompositePtr m_scene;
    CIvfCameraPtr m_camera;
    CIvfLightPtr m_light;
```

To make deallocation of the Ivf++ objects easier we use smart pointers for all the member variables, indicated by the `Ptr` extension in the class declarations. Instantiation of the scene graph node and other objects are done in the `onInit` method. First, the root scene graph object is created:

```
m_scene = new CIvfComposite();
```

As seen in the instantiation code, using a smart pointer is as simple as using a normal pointer variable in C++.

Next a simple cube object is created with its associated material attribute.

```
CIvfMaterialPtr material = new CIvfMaterial();
material->setDiffuseColor(1.0f, 0.0f, 0.0f, 1.0f);
material->setSpecularColor(1.0f, 1.0f, 1.0f, 1.0f);

CIvfCubePtr cube = new CIvfCube();
cube->setMaterial(material);
```

Adding objects to a composite object is done using the `addChild` method.

```
m_scene->addChild(cube);
```

To view the scene a view transform and a perspective transform is needed. This is handled in Ivf++ by the `CIvfCamera` class.

```
m_camera = new CIvfCamera();
m_camera->setPerspective(45.0, 0.1, 20.0);
m_camera->setPosition(3.0, 3.0, 3.0);
```

In the code above the parameters for the perspective transform is set by the `setPerspective`. The first parameter is the field of view, the second and third parameters set the near and far clipping planes.

To be able to see any objects in the scene graph, lighting must also be enabled. Lighting in Ivf++ is handled through the `CIvfLighting` singleton class. This class maintains a set of lights corresponding to the lights available in OpenGL. Lighting is setup using the following code:

```
CIvfLightingPtr lighting = CIvfLighting::getInstance();
m_light = lighting->getLight(0);
m_light->setLightPosition(1.0, 1.0, 1.0, 0.0);
m_light->setAmbientColor(0.2f, 0.2f, 0.2f, 1.0f);
m_light->enable();
```

Viewport and camera must be adjusted when the window size changes, this is done in the `onResize` event method.

```
m_camera->setViewPort(width, height);
m_camera->initialize();
```

To render the scene graph, the code in the `onRender` event method must be modified to render the scene graph, lighting and view transforms.

```
m_light->render();
m_camera->render();
m_scene->render();
```

The ordering in the code above is important. A camera must always be rendered before the scene graph. Rendering lights can be done before the camera or after, producing different effects. Rendering the light before the camera, will have the effect of having the light attached to the camera. Rendering the light after the camera has the effect of placing the light relative to the scene graph coordinate system. The complete code is found in the following listing:

Listing 2: Ivf++ scene graph rendering

```
#include <ivfui/IvfApplication.h>
#include <ivfui/IvfWindow.h>

#include <ivf/IvfComposite.h>
#include <ivf/IvfCamera.h>
#include <ivf/IvfMaterial.h>
#include <ivf/IvfCube.h>
#include <ivf/IvfLighting.h>
#include <ivf/IvfLight.h>

IvfSmartPointer(CExampleWindow);

class CExampleWindow: public CIvfWindow,
    CIvfInitEvent,
    CIvfResizeEvent,
    CIvfRenderEvent {
private:
    CIvfCompositePtr m_scene;
    CIvfCameraPtr m_camera;
    CIvfLightPtr m_light;
public:
    CExampleWindow(int X, int Y, int W, int H);
```

A-31

```
   virtual void onInit(int width , int height);
   virtual void onResize(int width , int height);
   virtual void onRender();
};

CExampleWindow::CExampleWindow(int X, int Y, int W, int H)
:CIvfWindow(X, Y, W, H)
{
   addInitEvent(this);
   addResizeEvent(this);
   addRenderEvent(this);
}

void CExampleWindow::onInit(int width , int height)
{
   m_scene = new CIvfComposite();

   CIvfMaterialPtr material = new CIvfMaterial();
   material->setDiffuseColor(1.0f , 0.0f , 0.0f , 1.0f);
   material->setSpecularColor(1.0f , 1.0f , 1.0f , 1.0f);

   CIvfCubePtr cube = new CIvfCube();
   cube->setMaterial(material);

   m_scene->addChild(cube);

   m_camera = new CIvfCamera();
   m_camera->setPerspective(45.0, 0.1, 20.0);
   m_camera->setPosition(3.0, 3.0, 3.0);

   CIvfLightingPtr lighting = CIvfLighting::getInstance();
   m_light = lighting->getLight(0);
   m_light->setLightPosition(1.0, 1.0, 1.0, 0.0);
   m_light->setAmbientColor(0.2f , 0.2f , 0.2f , 1.0f);
   m_light->enable();
}

void CExampleWindow::onResize(int width , int height)
{
   m_camera->setViewPort(width , height);
   m_camera->initialize();
}

void CExampleWindow::onRender()
{
   m_light->render();
   m_camera->render();
   m_scene->render();
}

int main(int argc , char **argv)
{
   CIvfApplicationPtr app = new CIvfApplication(IVF_DOUBLE|IVF_RGB);

   CExampleWindowPtr window = new CExampleWindow(0, 0, 512, 512);
   window->setWindowTitle("Ivf++_OpenGL_application");
   window->show();

   app->run();

   return 0;
}
```

Listing 2: Ivf++ scene graph rendering

3.1   Using the Ivf++ CIvfScene class

To make it easier to use a scene graph, Ivf++ includes a special class CIvfScene for handling
lighting, viewing and rendering. To change the previous example to use the CIvfScene class
the include directives for the CIvfComposite is changed to the following:

```
#include <ivf/IvfScene.h>        // Changed
#include <ivf/IvfCamera.h>
#include <ivf/IvfMaterial.h>
#include <ivf/IvfCube.h>
#include <ivf/IvfLighting.h>
#include <ivf/IvfLight.h>
```

In the class declaration the `CIvfCompositePtr` is changed to `CIvfScenePtr`. The `m_camera` and `m_light` member variables can be removed because the `CIvfScene` class will handle the camera and the light rendering.

```
class CExampleWindow: public CIvfWindow,
    CIvfInitEvent,
    CIvfResizeEvent,
    CIvfRenderEvent {
private:
    CIvfScenePtr m_scene;
```

The `onInit` must be updated to instantiate a `CIvfScene` class instead of a `CIvfComposite` class.

```
m_scene = new CIvfScene();
```

The camera is instantiated locally and assigned to the `m_scene` object.

```
CIvfCameraPtr camera = new CIvfCamera();
camera->setPerspective(45.0, 0.1, 20.0);
camera->setPosition(3.0, 3.0, 3.0);

m_scene->setCamera(camera);
```

Rendering of the lights are now handled by the `CIvfScene` class. Configuration of the light still has to be done, but it can be done locally.

```
CIvfLightingPtr lighting = CIvfLighting::getInstance();
CIvfLightPtr light = lighting->getLight(0);
light->setLightPosition(1.0, 1.0, 1.0, 0.0);
light->setAmbientColor(0.2f, 0.2f, 0.2f, 1.0f);
light->enable();
```

The `CIvfScene` class implements a method, `doResize` for updating camera and viewport, which must be called from the `onResize` method.

```
void CExampleWindow::onResize(int width, int height)
{
    m_scene->doResize(width, height);
}
```

The `onRender` event method now becomes even simpler. Only a single call to the `CIvfScene` object is needed to render the scene.

```
void CExampleWindow::onRender()
{
    m_scene->render();
}
```

The complete example is found in the following listing:

Listing 3: Ivf++ scene graph rendering using the `CIvfScene` class

```cpp
#include <ivfui/IvfApplication.h>
#include <ivfui/IvfWindow.h>

#include <ivf/IvfScene.h>
#include <ivf/IvfCamera.h>
#include <ivf/IvfMaterial.h>
#include <ivf/IvfCube.h>
#include <ivf/IvfLighting.h>
#include <ivf/IvfLight.h>

IvfSmartPointer(CExampleWindow);

class CExampleWindow: public CIvfWindow,
    CIvfInitEvent,
    CIvfResizeEvent,
    CIvfRenderEvent {
private:
    CIvfScenePtr m_scene;
public:
    CExampleWindow(int X, int Y, int W, int H);

    virtual void onInit(int width, int height);
    virtual void onResize(int width, int height);
    virtual void onRender();
};

CExampleWindow::CExampleWindow(int X, int Y, int W, int H)
: CIvfWindow(X, Y, W, H)
{
    addInitEvent(this);
    addResizeEvent(this);
    addRenderEvent(this);
}

void CExampleWindow::onInit(int width, int height)
{
    m_scene = new CIvfScene();

    CIvfMaterialPtr material = new CIvfMaterial();
    material->setDiffuseColor(1.0f, 0.0f, 0.0f, 1.0f);
    material->setSpecularColor(1.0f, 1.0f, 1.0f, 1.0f);

    CIvfCubePtr cube = new CIvfCube();
    cube->setMaterial(material);

    m_scene->addChild(cube);

    CIvfCameraPtr camera = new CIvfCamera();
    camera->setPerspective(45.0, 0.1, 20.0);
    camera->setPosition(3.0, 3.0, 3.0);

    m_scene->setCamera(camera);

    CIvfLightingPtr lighting = CIvfLighting::getInstance();
    CIvfLightPtr light = lighting->getLight(0);
    light->setLightPosition(1.0, 1.0, 1.0, 0.0);
    light->setAmbientColor(0.2f, 0.2f, 0.2f, 1.0f);
    light->enable();
}

void CExampleWindow::onResize(int width, int height)
{
    m_scene->doResize(width, height);
}

void CExampleWindow::onRender()
{
    m_scene->render();
}

int main(int argc, char **argv)
{
```

```
CIvfApplicationPtr app = new CIvfApplication(IVF_DOUBLE|IVF_RGB);

CExampleWindowPtr window = new CExampleWindow(0, 0, 512, 512);
window->setWindowTitle("Ivf++_OpenGL_application");
window->show();

app->run();

return 0;
}
```

Listing 3: Ivf++ scene graph rendering using the `CIvfScene` class

4   Using event handlers

A new concept introduced from version 0.9.x in Ivf++ is event handlers. Event handlers are classes implementing a functionality that integrates with events in the `CIvfWidgetBase` derived classes, such as the `CIvfWindow` class used in the examples in this article. Using event handlers can minimise the event handling code in an application, enabling a user to focus on the visualisation code. The relationships between the handler and widget base class is shown in figure 3.
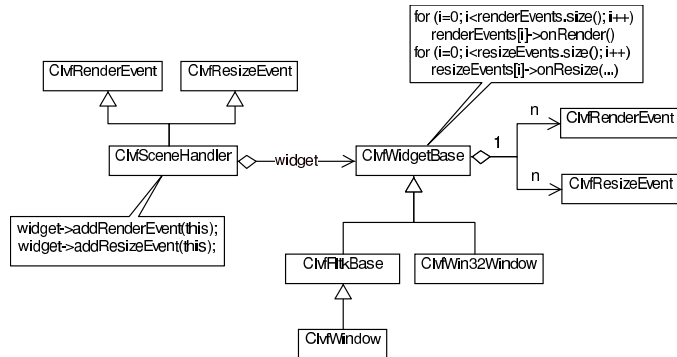
Figure 3: Event handler concepts

The scene graph example in section 3.1 will be modified to use handler classes for scene rendering and mouse viewing.

Handler classes are located in the `ivfwidget` library. In this example the `CIvfSceneHandler` and `CIvfMouseViewHandler` classes will be used, so the following includes are added:

```
#include <ivfwidget/IvfSceneHandler.h>
#include <ivfwidget/IvfMouseViewHandler.h>
```

In the class declaration, events for resizing and rendering are removed. These events are handled automatically by the handler classes. The `CIvfInitEvent` event is still needed to setup the scene graph. The member variables for the handler classes `m_sceneHandler` and `m_mouseViewHandler` are added.

```
class CExampleWindow: public CIvfWindow,
        CIvfInitEvent
{
private:
        CIvfScenePtr m_scene;
        CIvfSceneHandlerPtr m_sceneHandler;
        CIvfMouseViewHandlerPtr m_mouseViewHandler:
```

```
    public:
            CExampleWindow(int X, int Y, int W, int H);

            virtual void onInit(int width, int height);
    };
```

The constructor must also be modified, removing the resize and render events. The constructor becomes:

```
    CExampleWindow::CExampleWindow(int X, int Y, int W, int H
        )
    :CIvfWindow(X, Y, W, H)
    {
            addInitEvent(this);
    }
```

In the onInit event method, the handler classes must be instantiated. First the CIvfScene class is instantiated. This class will automatically handle both rendering and resizing of the window. The constructor needs a reference to a CIvfWidget derived instance, in this case the CIvfExampleWindow class (this) and a reference to a CIvfScene instance.

```
    m_sceneHandler = new CIvfSceneHandler(this, m_scene);
```

The CIvfMouseViewHandler class handles mouse input for controlling the current view. The constructor needs a reference to a CIvfWidget derived instance, in this case the CIvfExamp class and a reference to a CIvfCamera instance for controlling the camera.

```
    m_mouseViewHandler = new CIvfMouseViewHandler(this,
        camera);
```

A handler in Ivf++ can be active or inactive. When a handler is created it is default set to active, so no extra code is needed for the handler to receive events. To change handler state the activate and deactivate methods can be used. The last step is to remove the event methods onResize and onRender in the class implementation. The application is now finished, containing only one event for initialising the scene graph.

Other handler classes in Ivf++ are:

library	Description
CIvfInteractionHandler	Interaction with 3D GUI controls.
CIvfCoordinateInputHandler	3D Coordinate input using the mouse.
CIvfSelectionHandler	Shape selection.
CIvfFlyHandler	Fly view handler.

Table 2: Other handler classes

The complete example is found in the following listing:

Listing 4: Using handler classes in Ivf++

```
#include <ivfui/IvfApplication.h>
#include <ivfui/IvfWindow.h>

#include <ivf/IvfScene.h>
#include <ivf/IvfCamera.h>
#include <ivf/IvfMaterial.h>
#include <ivf/IvfCube.h>
#include <ivf/IvfLighting.h>
#include <ivf/IvfLight.h>

#include <ivfwidget/IvfSceneHandler.h>
```

```cpp
#include <ivfwidget/IvfMouseViewHandler.h>

IvfSmartPointer(CExampleWindow);

class CExampleWindow: public CIvfWindow,
    CIvfInitEvent
{
private:
    CIvfScenePtr m_scene;
    CIvfSceneHandlerPtr m_sceneHandler;
    CIvfMouseViewHandlerPtr m_mouseViewHandler;
public:
    CExampleWindow(int X, int Y, int W, int H);

    virtual void onInit(int width, int height);
};

CExampleWindow::CExampleWindow(int X, int Y, int W, int H)
:CIvfWindow(X, Y, W, H)
{
    addInitEvent(this);
}

void CExampleWindow::onInit(int width, int height)
{
    m_scene = new CIvfScene();

    CIvfMaterialPtr material = new CIvfMaterial();
    material->setDiffuseColor(1.0f, 0.0f, 0.0f, 1.0f);
    material->setSpecularColor(1.0f, 1.0f, 1.0f, 1.0f);

    CIvfCubePtr cube = new CIvfCube();
    cube->setMaterial(material);

    m_scene->addChild(cube);

    CIvfCameraPtr camera = new CIvfCamera();
    camera->setPerspective(45.0, 0.1, 20.0);
    camera->setPosition(3.0, 3.0, 3.0);

    m_scene->setCamera(camera);

    CIvfLightingPtr lighting = CIvfLighting::getInstance();
    CIvfLightPtr light = lighting->getLight(0);
    light->setLightPosition(1.0, 1.0, 1.0, 0.0);
    light->setAmbientColor(0.2f, 0.2f, 0.2f, 1.0f);
    light->enable();

    m_sceneHandler = new CIvfSceneHandler(this, m_scene);
    m_mouseViewHandler = new CIvfMouseViewHandler(this, camera);
}

int main(int argc, char **argv)
{
    CIvfApplicationPtr app = new CIvfApplication(IVF_DOUBLE|IVF_RGB);

    CExampleWindowPtr window = new CExampleWindow(0, 0, 512, 512);
    window->setWindowTitle("Ivf++_OpenGL_application");
    window->show();

    app->run();

    return 0;
}
```

Listing 4: Using handler classes in Ivf++

5 Extending the library

Many class libraries often have a special abstraction layer hiding the lower level libraries or software rendering implementations. This approach is flexible when supporting DirectX [1]

or OpenGL [9] in the same class library. Extending such a library with custom rendering code can only be done through the abstraction layer, thus reducing the amount of functionality available. Ivf++ does not have this problem, because it is designed to support and enable the use of OpenGL in an easy way. The class library serves as a framework supporting the use of OpenGL code directly in the classes.

There are three main ways of extending the library with rendering code:

1. Deriving `CIvfShape` derived classes with custom OpenGL rendering code.

2. Deriving `CIvfComposite` derived classes composed of other Ivf++ shape classes.

3. Deriving `CIvfGLPrimitive` derived classes creating topology based geometry using the functions in the base class.

To aid in the creation of extension classes, Ivf++ includes a simple class generator, `ivfclassgen` for this purpose. `ivfclassgen` is a simple command line utility written in Python [10] which generates skeleton include and implementation files for the 3 main extension types. Giving the command `ivfclassgen` without any parameters lists the available command options:

```
Ivf Classgen 0.1 - Ivf++ Class Template Generator
Copyright (C) 2003 Division of Structural Mechanics
Usage:
        ivfclassgen classtype [classname]

values for classtype are:

        shape           Shape derived class
        quadset         QuadSet derived class
        composite       Composite derived class
```

The first option specifies the type of class generated. The last options specifies the name of the class.

5.1 Creating `CIvfShape` derived classes

The most general way of extending Ivf++ is to create `CIvfShape` derived classes. The `CIvfShape` supports positioning, material, textures and state handling in OpenGL. To implement OpenGL rendering in the class the protected virtual method `createGeometry` must be overridden. In the `createGeometry` method any OpenGL based rendering code is placed.

Rendering code in the `CIvfShape` should generate the geometry around the origin $(0, 0, 0)$, otherwise the built in position code will not be intuitive to use. Material handling can be left out if not some kind of vertex colouring scheme is used. The `CIvfShape` class will automatically apply material parameters before the `createGeometry` method is called. If textures are used together with the extension class, texture coordinates must be specified using `glTexCoord2` methods.

To illustrate the process of creating an extension class, a simple class `CIvfShapeExample` will be implemented. Skeleton source code for the class defintion and implementation is generated using the `ivfclassgen` tool. The following commands are given:

```
        ivfclassgen shape IvfShapeExample
```

The files `IvfShapeExample.h` and `IvfShapeExample.cpp` are generated. The `ivfclassgen` tool automatically adds the neccesary class methods and also adds a smart pointer definition and support for the Ivf++ run time class information system. The generated code for `IvfShapeExample.h` is shown in the following listing:

```
#ifndef _CIvfShapeExample_h_
#define _CIvfShapeExample_h_
```

```
#include <ivf/IvfShape.h>

IvfSmartPointer(CIvfShapeExample);

class CIvfShapeExample : public CIvfShape {
private:

public:
        CIvfShapeExample();

        IvfClassInfo("IvfShapeExample", CIvfShape);

protected:
        virtual void createGeometry();
};

#endif
```

To implement the new shape, code for rendering must be added in the `createGeometry` method. The `ivfclassgen` tool creates some simple rendering code to illustrate the concepts, as shown in the following code:

```
#include "IvfShapeExample.h"

CIvfShapeExample::CIvfShapeExample()
{
        // Add construction code here
}

void CIvfShapeExample::createGeometry()
{
        // Rendering code

        glBegin(GL_QUADS);
        glNormal3d( 0.0, 0.0, 1.0);
        glVertex3d( 1.0, 1.0, 0.0);
        glVertex3d(-1.0, 1.0, 0.0);
        glVertex3d(-1.0,-1.0, 0.0);
        glVertex3d( 1.0,-1.0, 0.0);
        glEnd();
}
```

The implemented class can automatically be used with Ivf++ in the same way as any other Ivf++ class. To illustrate this, the cube in the previous examples are replaced with the new `CIvfShapeExample` class. First the include for the new class is added:

```
#include "IvfShapeExample.h"
```

In the `onInit` the cube instantiation code is replaced with code for instantiating the new class:

```
CIvfShapeExamplePtr newShape = new CIvfShapeExample();
newShape->setMaterial(material);

m_scene->addChild(newShape);
```

As shown in the code above the normal Ivf++ method `setMaterial` is used to assign a material that will be used when rendering the object. Running the example will produce a single red plane.

5.2 Creating `CIvfGLPrimitive` derived classes

To implement advanced geometry in OpenGL often involves the handling of coordinate arrays, index arrays and normal arrays. To aid in this process Ivf++ supports the handling of indicies and coordinates through the `CIvfGLPrimitive` base class. From this class Ivf++ several classes for the OpenGL rendering primitives are derived, see figure 4.
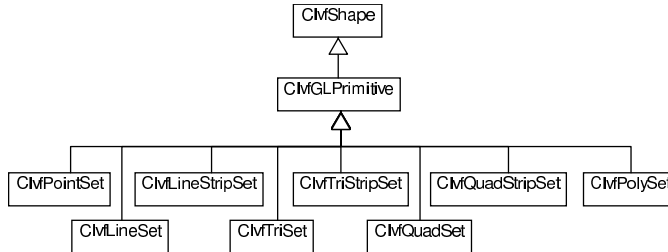

Figure 4: `CIvfGLPrimitive` derived classes in Ivf++

Creating classes based on the `CIvfGLPrimitive` base class can be done either by directly deriving from the `CIvfGLPrimitive` or from the separate Ivf++ primitive classes. Deriving directly from the `CIvfGLPrimitive` class will also require implementing the `createGeometry` for rendering the geometry using OpenGL calls. To derive from the Ivf++ primitive base classes require defining the geometry using the methods in the `CIvfGLPrimitive` class and implementing routines for changing geometry dimensions. To illustrate this a simple `CIvfQuadSet` derived class `CIvfQuadSetExample` will be implemented. The class renders a single quadrilateral with properties for width and height. The class definition is shown below:

```
#ifndef _CIvfQuadSetExample_h_
#define _CIvfQuadSetExample_h_

#include <ivf/IvfQuadSet.h>

IvfSmartPointer(CIvfQuadSetExample);

class CIvfQuadSetExample: public CIvfQuadSet {
private:
        double m_width;
        double m_height;

        void update();
public:
        CIvfQuadSetExample();

        IvfClassInfo("IvfQuadSetExample", CIvfQuadSet);

        void setSize(double width, double height);
};

#endif
```

A special routine `update` is added, for updating the geometry when the width and height properties are changed. The initial topology of the geometry can not be changed in this class, so the geometry definition is done in the class constructor. First default values for the width and height are set

```
CIvfQuadSetExample :: CIvfQuadSetExample ()
{
        m_width = 1.0;
        m_height = 1.0;
```

The geometry is centered around $(0, 0, 0)$ so 3 offset variables are defined.

```
double ox = −m_width /2.0;
double oy = −m_height /2.0;
double oz = 0.0;
```

Coordinates are added using the `addCoord` from the `CIvfGLPrimitive`.

```
addCoord(ox, oy, oz);
addCoord(ox + m_width, oy, oz);
addCoord(ox + m_width, oy + m_height, oz);
addCoord(ox, oy + m_height, oz);
```

To support textures the class must supply texture coordinates. Adding texture coordinates is done using the `addTextureCoord` method. This example implements a very simple texture mapping scheme, as shown in figure 5.
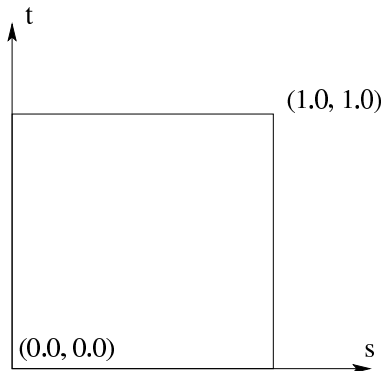

Figure 5: Texture coordinates in the `CIvfQuadSetExample` class

The texture coordinates for the described mapping is added with the following code:

```
addTextureCoord (0.0, 0.0);
addTextureCoord (1.0, 0.0);
addTextureCoord (1.0, 1.0);
addTextureCoord (0.0, 1.0);
```

Until now no connectivity data has not been defined. Connectivity in Ivf++ is handled using the `CIvfIndex` class. The `CIvfIndex` class maintains an array of indices (long) pointing to coordinates, texture coordinates or vertex normals. `CIvfGLPrimitive` derived classes can have one or more `CIvfIndex`, each instance representing a `glBegin`/`glEnd` pair. In this example 2 `CIvfIndex` indexes are needed, one for the vertices and one for the texture coordinates. Using a `CIvfIndex` class is simple, indexes are added with the `add` method. The `CIvfIndex` instance for the coordinates is added to the `CIvfGLPrimitive` class with the `addCoordIndex` method.

```
CIvfIndexPtr idx;

idx = new CIvfIndex();
idx->add(1, 2, 3, 0);

addCoordIndex(idx);
```

Texture indices are added in the same way.

```
idx = new CIvfIndex();
idx->add(0, 1, 2, 3);

addTextureIndex(idx);
```

Changing the size of the quad using the `setSize`, requires changing the defined geometry. This is done in the `update` routine. The `CIvfGLPrimitive` provides the method `setCoord` for changing the coordinates in the coordinate array. The `update` method becomes as followes:

```
void CIvfQuadSetExample::update()
{
        double ox = -m_width/2.0;
        double oy = -m_height/2.0;
        double oz = 0.0;

        setCoord(0, ox, oy, oz);
        setCoord(1, ox + m_width, oy, oz);
        setCoord(2, ox + m_width, oy, oz + m_height);
        setCoord(3, ox, oy, oz + m_height);
}
```

Changing texture coordinates is done in the same way using the `setTextureCoord` method.

5.3 Creating `CIvfComposite` derived classes

The easiest way of extending Ivf++ is to create composite objects consisting of existing Ivf++ classes. These types of classes can be derived from the `CIvfComposite` class. Objects contained in the composite are created and added in the class constructor. The class definition can look as follows:

```
#ifndef _CIvfCompositeExample_h_
#define _CIvfCompositeExample_h_

#include <ivf/IvfComposite.h>

IvfSmartPointer(CIvfCompositeExample);

class CIvfCompositeExample: public CIvfComposite {
private:

public:
        CIvfCompositeExample();

        IvfClassInfo("IvfCompositeExample", CIvfComposite
            );
};

#endif
```

The example above only includes a declaration for the class constructor. If the class implements dynamic geometry, access methods and member variables for the objects has to be added in the class definition.

Objects created in the class constructor are added using the `addChild` method of the `CIvfComposite` class.

```cpp
#include "IvfCompositeExample.h"

#include <ivf/IvfCube.h>
#include <ivf/IvfSphere.h>

CIvfCompositeExample::CIvfCompositeExample()
{
        // Add construction code here

        CIvfCubePtr cube = new CIvfCube();

        this->addChild(cube);

        CIvfSpherePtr sphere = new CIvfSphere();
        sphere->setRadius(0.5);
        sphere->setPosition(0.0, 0.5, 0.0);

        this->addChild(sphere);
}
```

The code above creates a 1×1 cube located at $(0,0,0)$. A sphere with $r = 0.5$ is placed with the centre at $(0, 0.5, 0)$.

6   Conlusions

Ivf++ is an easy to use framework, supporting OpenGL application development. The current version 0.9.x includes approximately 240 classes. Integration with different graphical user interface toolkits is an important factor in the development of Ivf++, and the library presently integrates with FLTK [3], MFC [8] and native Windows applications.

Another important aspect of the library is to support the OpenGL library not by completely encapsulating it behind an abstraction layer, but to support the direct use of it in the Ivf++ classes. Software rendering is supported through the MesaGL [7] library and on Microsoft Windows using the fallback method available in the OpenGL implementation on this platform.

Ivf++ can also effectively be used as a rapid application development system for OpenGL applications. Ivf++ includes a simple application framework, enabling the creation of platform independant OpenGL applications, without any platform specific code.

Documentation is an important aspect often neglected. Most classes in Ivf++ are fully documented using the Doxygen [2] documentation tool. Doxygen produces reference documenation in HTML, RTF or Latex. Ivf++ also comes with a 200 page user's guide [6].

The driving force behind the development of this library has been the feedback and the many downloads of the library on the internet [1]. The library is released under a LGPL license enabling the use of the library in open source applications as well as commercial propriatary applications.

References

[1] Microsoft DirectX - Multimedia technology for Windows-based gaming and
    entertainment, http://www.microsoft.com/windows/directx/default.aspx, 2003

---

f++ has been downloaded 12000 times since 2000