An Object-Oriented Interactive System for Scientific Simulations: Design and Applications

A.C. Telea and C.W.A.M. van Overveld

Eindhoven University of Technology, Department of Mathematics and Computing Science, Den Dolech 2, 5600 MB, Eindhoven, The Netherlands
E-mail: alext@win.tue.nl, wsinkvo@win.tue.nl

Abstract. Better insight in complex physical processes requires integration of scientific visualization and numerical simulation in a single interactive framework. This paper presents an object-oriented environment which combines the tasks of numerical simulation, visualization, simulation specification, run-time monitoring and steering.

We first review the different existing approaches to the above tasks and outline their relative limitations. Next, we present a model for a framework which attempts to provide a general approach to the tasks of simulation specification and steering in an object-oriented manner. An implementation of the framework is described.

We have built an object-oriented library for finite element computations and integrated it into the simulation system. An example in which our system has been used to solve a practical engineering problem illustrates the combination of object-oriented numerics and interactivity.

1 Introduction

Scientific visualization has been massively used in numerous fields in order to get an insight into various complex physical processes. Interactivity, seen as the ability of the user to examine and modify the universe she observes, has become a critical requirement of simulation and visualization tools, whether they represent numerical simulations of physical processes, computer animations or virtual reality world models. Interaction can come in the form of changing visualization parameters, in which case we have a *visualization system*, or changing parameters of the modelled process, in which case we have a more general *simulation system*.

Modelling capabilities represent another essential requirement for simulation systems which should provide an easy, natural way to specify the simulated universe in terms of high-level, modular entities which closely parallel the concepts of the real problem to be described.

Although many simulation and visualization systems exist, few of them provide a generic framework combining the abstractions required for modelling complex physically-based or virtual universes together with full interaction freedom with all the simulation parameters.

We have attempted to answer to the above requirements by designing a general purpose system for scientific simulations. The proposed system addresses the tasks of simulation specification and interactivity in a uniform manner, via a high-level object-oriented user interface.

The organization of this article is as follows. Section 2 presents an overview of the existing types of simulation frameworks and outlines their relative limitations. Section 3 presents the conceptual model and the design of the proposed system. For this we firstly introduce the concepts of dependency graphs and relationships in object-oriented programming. Then, we show how we combine object-oriented specifications with constraint management into a homogeneous, interactive environment. Although the presented simulation system is general purpose, we have concentrated on support for simulations using the finite element (FEM) method. Section 4 introduces a FEM simulation library that we have built and integrated in our system. Sections 6 and 7 illustrate the functionality of the FEM library by a couple of simulation examples. The last section discusses the directions of our ongoing research.

2 Previous Work

The simplest simulation frameworks come as libraries dedicated to a limited range of operations, such as geometrical modelling [1], linear algebra [2] or visualization [3]. Object-oriented libraries such as Diffpack [5] or LAPACK++ [6] provide a more abstract application programming interface (API) by which the user can represent simulation concepts as *objects*.

Specification of complex simulations can be however only partially done by such libraries. Besides modelling the simulation's entities by objects, the programmer should represent the *relationships* between these entities. For example, a numerical simulation can have many parameters depending on each other in complex ways. Such dependencies impose *constraints* on the time evolution of the parameters they involve. Since it is complicated and error-prone for the user to 'steer' such a simulation by explicitly changing all its parameters and maintaining the constraints, a *constraint management* mechanism is provided to specify and automatically enforce constraint relationships. A good example of a simulation library offering constraint management is OpenInventor [4].

Adding *interactivity* to simulation systems takes yet another step in modelling reality. While some systems allow only the monitoring of time dependent data, interactive steering systems practically integrate numerical computations and visualization in one tool, which can monitor but also interactively steer a running simulation. Haber and McNabb [7] and Marshall et al. [8] give a good survey of interactive simulation systems.

Dataflow systems like AVS [9], Khoros, Iris Explorer or apE provide some of the above features. The simulation is interactively specified by means of a graphical user interface (GUI) which allows connecting various computational modules in a directed graph, called a flow network. While the simulation runs, data flows from its source through modules which perform various operations on it up to the modules which perform the effective visualization.

Although powerful, most dataflow systems have a series of limitations. Explicit (by value) data transfer between modules is mainly used, which is time and memory consuming in case of large data sets. Secondly, most dataflow systems use purely procedural (state-less) modules, which often don't offer the abstraction level required for modelling complex processes (e.g. simulations described by coupled partial differential equations). Moreover, the dataflow model can express constraints only by pipelining modules in a flow network. We would like a more abstract, possibly object-oriented way of specifying relationships between the entities involved in a simulation. Finally, the integration of many existing object-oriented libraries in a dataflow system raises serious problems. These are partly caused by the system's interface inability to deal with object-oriented entities and relationships, partly by the dynamic and interactive nature of a simulation which may conflict with the library's design philosophy.

The simulation framework we propose uses an object-oriented approach to simulation specification, user interaction and visualization. Constraint specification can be done either in the manner provided by dataflow systems or in a more powerful, objectual way, via an object-oriented GUI of a special design. The next section presents these issues.

3 Conceptual Model and Design of the Simulation System

We shall firstly present a conceptual model of a simulation, which is used by the software system as a simulation basic representation form. Atop of this basic model, the system uses a more sophisticated specification paradigm, which we designed by combining the object-oriented and constraint specification policies. We present the advantages of the combined specification approach and show how it is mapped on the basic representation.

3.1 Conceptual Model

The conceptual model is based on the notion of *state*, defined as the set of parameters that fully characterize the system at a given time instant. These parameters (also called state variables) can model physical quantities of the system (e.g. simulation time, velocity of a body in an animation, pressure and vorticity of a fluid in a flow simulation) but can also be parameters of the visualization system monitoring a simulation or the convergence rate of a linear algebra solver.

A second concept is the *dependency* or law. If a state parameter b depends on a state parameter a, then whenever a changes, b must change as well in order to maintain a constitutive law of the process. For example, if the position x of a body depends on time t, this can be expressed by a law $x = x(t)$. More complex laws can express the dependency of the temperature in each point of a body on the temperature on its boundary by means of a diffusion partial differential equation (PDE).

The set of all state variables and dependencies of a simulation constitutes the system's dependency graph.

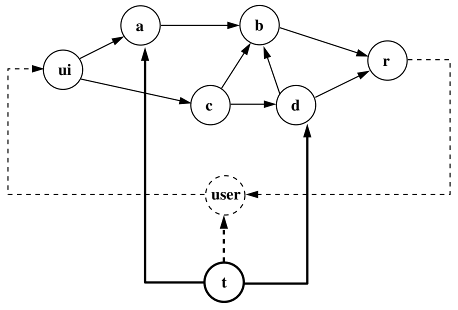


Fig. 1. Dependency graph illustrating state variables and laws. A directed arc (e.g. from node a to b) represents a dependency of the form $b = b(a)$

Figure 1 depicts such a dependency graph. A dependency $b = b(a)$ can be implemented as a a functional or procedural module which evaluates the state parameter b given the parameter a. The system's parameters directly controlled by the user (i.e. the user interface) are modelled by the state parameter ui, while the state parameter r represents all parameters directly observable by the user (e.g. graphics or numerical output). An interesting feature of the model is that the human user can be represented by a parameter determining the ui parameter and depending on the system's r parameter via a complex feedback law. Time t is the only independent parameter since all other parameters are, directly or indirectly, seen as functions of time. The human user's implicit dependency on time is represented by a dotted arrow.

A dependency graph is a complete specification of an arbitrarily complex simulation. Indeed, all entities in the simulation are completely characterized by their state parameters, the system's behaviour in time is described by the laws or dependencies between these parameters and user interaction with the system is described in terms of the ui and r state parameters. When a state parameter changes, the system traverses the dependency graph from the corresponding node and uses the existing laws to evaluate all the parameters encountered during the traversal.

Although the above model is general and very convenient for expressing and enforcing dependencies, the description of a simulation in terms of state variables is a too low level modelling paradigm. A different approach views a simulation in terms of *objects*, which practically group subsets of state variables and allow the user to treat them as a whole by means of specialized methods. Although objects elegantly model simulation concepts, they are unable to *directly* express and enforce dependencies between their parameters.

We have combined the two paradigms into one system. Firstly, we use *objects* (implemented in the C++ programming language) to model the entities of our simulation. Then, we express *dependencies* between these objects in order to describe the laws of our simulation. Finally, we provide a mechanism which automatically enforces these dependencies.

Interactivity introduces a third degree of freedom in the system's design. The user should be able to create, destroy, modify and examine objects while the simulation is running. We designed an object-oriented graphical user interface (GUI) which associates an objectual 'widget' to each class in the system. The GUI also offers an object-oriented means to managing dependencies between objects.

We have kept the three design issues involved in the system (object-oriented programming, constraint programming and interactivity) orthogonal. That is, simulation objects can be designed independently on the constraint specification mechanism (see [12] about a discussion on the problems which arise) *and* on the interaction paradigm the system uses. Firstly, this allows us to use class libraries which have been designed independently on our simulation system. Constraint management can be transparently added to such classes without having to reprogram them. The same is true for the object-oriented GUI widgets associated to the classes. In conclusion, we can change or upgrade any component of the system without changing the other two.

3.2 Design of an Object-Oriented Simulation System

The simulation system consists of three main parts, implementing the three main functionalities previously outlined: the object manager, the dependency manager and the interaction manager (see Fig. 2 for an overview). All these parts communicate together by sharing the dependency graph simulation description.

The *object manager* is the interface between the system and the simulation-specific class libraries. It keeps a registry of all the classes known by the simulation system and allows the user to dynamically create, destroy and copy instances of any of these classes at run-time via a GUI. Instances are referenced by names, exactly like objects in the C++ language. Since the object manager does not use any information on simulation classes besides their names, any application-specific class library can be easily integrated into the system.

The *dependency manager* offers a way to interactively express dependencies between simulation objects. Dependencies are represented by objects which are similar to the computational modules encountered in dataflow systems. Besides such dependencies, the system introduces the capability

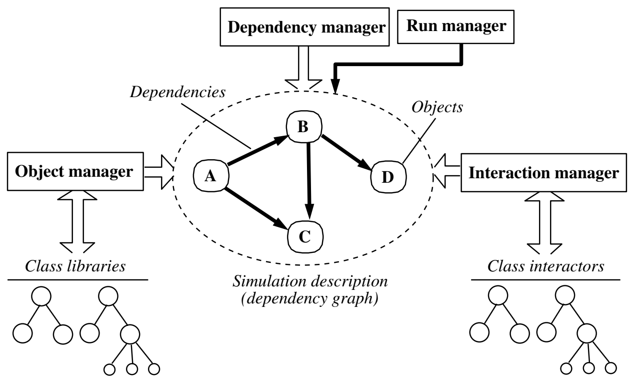


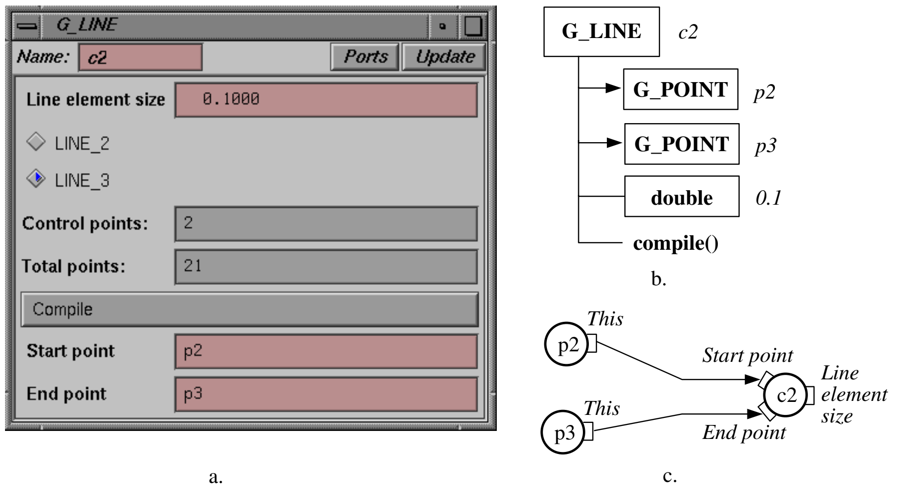
Fig. 2. Overview of the object-oriented simulation system.

to interactively build 'has-a' and 'uses-a' relationships which are specific to object-oriented programming.

A class **A** *has-a* class **B** if **A** has **B** as a member. Similarly, class **A** *uses-a* class **B** if **A** has a pointer to **B** as a member (Fig. 3 b). Together with inheritance (the *is-a* relationship), the *has-a* and *uses-a* relationships are the fundamental tools for expressing data dependencies in object-oriented class libraries. While *is-a* is a 'static' relationship (class hierarchies are constructed at compile-time), the *has-a* and *uses-a* relationships are established at run-time.

The dependency manager allows the user to create or destroy *has-a* and *uses-a* relationships between classes in an interactive way. The objectual *has-a* and *uses-a* relationships are automatically translated into a low level dependency graph similar to the one presented in the previous section. The nodes of this graph are the simulation objects and the arcs are the relationships between objects. A node has a set of typed *ports* which represent the publicly accessible data members of its object (i.e. its state). An arc is a connection between two ports of compatible types, meaning that there is a dependency between the data members corresponding to the ports (Fig. 3 c). Arcs can express both *has-a* (by-value) data dependencies (data is copied from one port to another to enforce the equality constraint) or *uses-a* (by-reference) dependencies (the two ports share the same physical data member). The user's attempts to create invalid relationships are prevented by an object-oriented run-time type checking component. If the user wishes however to have total control over the existing constraints, she can directly interact with the dependency graph and add or remove connections between ports in a way similar to the network management of the Oorange system [11].

A special component of the dependency manager, called the *run manager*, uses the dependency graph to propagate changes when an object is modified, thus ensuring the constraint satisfaction completely transparent to the user.



G_LINE

Name: c2

Line element size: 0.1000

LINE_2 [✓ LINE_3]

Control points: 2

Total points: 21

Start point: p2

End point: p3

Fig. 3. Objects, constraints and interactors. **a)** Interactor for a class *G_LINE*. **b)** Structure of class *G_LINE* (arrows are *uses* relationships, while lines are *has* relationships). **c)** Dependency graph for a *G_LINE* object (port names are written in italics).

The *interaction manager* offers a GUI to all objects in the system. The user can visually examine all the data members of any existing object, change their values (and see the effects the changes have on other objects if the changed items are involved in dependencies) or call the object's methods. Each class in the system has an *interactor*, which is a GUI widget displaying all public methods and data members of that class (Fig. 3 a). As sketched in Fig. 2, there is a one to one correspondence between simulation class hierarchies and interactor class hierarchies. This allows the programmer of a new simulation class which inherits from an existing class to rapidly derive an interactor from the one of the original class. Creating GUI interactors for existing C++ classes is facilitated by a set of predefined basic interactors for the fundamental C++ types (integers, floats, booleans, enumerations, typed pointers, arrays, etc).

To illustrate the relationship between classes, interactors and constraints, we shall use a very simple example of a line class (Figure 3. A class *G_LINE* represents a line as two references to two *G_POINT* objects being the line's end points, a double being the size of the element obtained when the line is meshed, a method **compile()** and some other less relevant data (Fig. 3 b). Three objects have been created: the line *c2* and the two end points *p1* and *p2*. We say that *c2 uses p1* and *p2 has* a double member (unnamed). Figure 3 a shows the interactor for class *G_LINE*, which allows visual control over all the line's members, e.g. change the start and end points or the line element size, call the **compile()** method or change the line object's name.

Figure 3 c) pictures the dependency graph, showing how *c1* depends on both *p1* and *p2*. The dependency graph is automatically modified as the user employs the GUI interactor to change, for example, the values of the **Start point** or **End point** widgets.

4 A Finite Elements Object-Oriented Library

Most of the existing finite element (FE) applications come in form of packages whose input is given as batch files and output is visualized in a postprocessing (post simulation analysis) phase. This separation of modelling, computations and result visualization limits the user's freedom to change and examine parameters of the FEM simulation to the preprocessing and postprocessing stages.

We have addressed the above limitation by designing an object-oriented library for finite element methods (FEM) and integrating it in the general-purpose simulation system previously presented. The library can be used also standalone, similarly to other object-oriented FE libraries such as Diffpack [5].

The combination of the OO FEM library with the simulation system creates a "virtual simulation and mathematical research" laboratory in which the problem specification, computation and result visualization tasks are fully interactive. End users of a simulation such as engineers can *steer* the ongoing process and monitor its evolution without quitting the simulation environment in order to redefine input files or recompile. Researchers can run FEM problems interactively and experiment with different numerical techniques or monitor error or convergence rates.

5 Structure of a Generic FE Simulation

Although functionally different, most FE simulations exhibit a similar generic structure. This structure and the FE library classes which occur in it are presented in the following.

A generic FE simulation has a dependency graph which consists of three main parts (Fig. 4), which correspond roughly to the modelling, computation and result visualization stages presented in the previous section. The main difference between such a dependency graph and the 'classical' 3-stages pipeline is that objects belonging to different simulation stages can be connected. This less clear separation between stages corresponds to an interleaving of modelling, computations and visualization and results in an increased interactivity.

A top-down presentation of the FE library classes found in each stage follows:

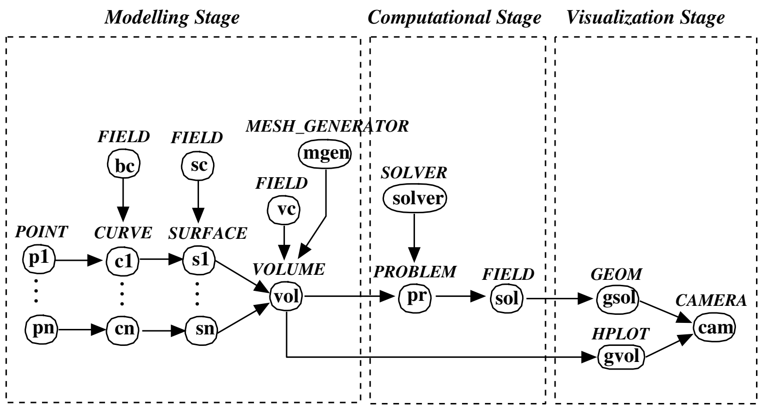


Fig. 4. Dependency graph for a generic FE simulation illustrating the three simulation stages with their respective **objects** and their *classes*.

5.1 Modelling Stage

Modelling comprises the specification of a geometric domain, boundary conditions and a PDE to be solved. These tasks are implemented by specific classes as follows:

Geometrical modelling: Geometries are specified by *POINT, CURVE, SURFACE* and *VOLUME* classes (Fig. 4, objects **p1..pn**, **c1..cn**, **s1..sn** and **vol**). Several *MESH_GENERATOR* classes (**mgen**) are available for discretization of the geometrical domain.

Boundary conditions: Boundary conditions of several types (e.g. Neumann, Dirichlet) are specified on the curves and/or surfaces of the geometrical domain (**bc**, **sc**). Besides its type, a condition uses also a *FIELD* object which represents an analytical or discrete function of position and gives the values prescribed for that condition.

PDE: The PDE is modelled by a *PROBLEM* class which contains specialized methods for that PDE type (e.g. building the stiffness matrix).

5.2 Computation Stage

The key class of this stage is the problem created during modelling (actually *PROBLEM* belongs to both the modelling and computation stages). In this stage, the problem is solved and its solution is written to a *FIELD* object **sol**. Other computation classes include several matrix types (sparse, diagonal, etc), iterative solvers (bi-conjugate gradient, generalized minimum residual, etc) and preconditioners, which overall form an OO linear algebra library similar to SparseLib++ or IML++ [14]. A number of 'low-level' FEM operations (building a stiffness matrix, renumbering schemes, etc) are implemented by private classes thus shielding the user from such technical details.

5.3 Visualization

This final stage comprises a set of classes which permit interactive visualization of the various data objects produced. Two types of classes are involved in this stage. The first type is used to *map* various data entities into objects which can be graphically represented (e.g. geometries *GEOM*, scalar and vector plots, Gouraud-shaded height plot classes *HPLOT*, etc). The second class type represents the *CAMERAs*, i.e. objects that graphically display the output of mappers and are interactively controlled by the user. Visualization classes are functionally similar to AVS's mapper and data output modules or to vtk's [10] rendering classes.

6 Example of Modelling a PDE: The Wave Equation

In this section we shall illustrate the FE library by a simple simulation based on a wave equation $\Delta u + c^2 \frac{\partial^2 u}{\partial t^2} = 0$ solved on a square domain with essential boundary conditions equal to zero. The wave is initiated by an exponential excitation function excitation $=$ height $*\exp^{-\frac{x^2+y^2}{width}}$ centered in some point on the geometrical domain. The simulation will generate the time dependent PDE solutions and present them to the user as 3D elevation plots. During the simulation, the user can change the position, amplitude and width parameters of the excitation and then superimpose the new excitation over the current PDE solution. The simulation will resemble dropping water droplets in a square bucket. The dependency graph for this simulation (Fig. 5) is presented in the following.

Geometry: The square's geometry is defined by four *POINT* objects **p1..p4**, four *LINE* specializations of *CURVE* **l1..l4** and a *SURFACE* **square** (the square itself). Note how the line objects share the point objects.

Boundary conditions: All boundaries have zero essential boundary conditions, specified by a specialization *CONST_FIELD* of the *FIELD* class (object **bc**) which models a constant field with zero value. This object is shared by all the curves.

Excitation: The excitation **excit** is described by a specialization *EXP_FIELD* of the *FIELD* class which models an exponential function. Note that **excit** depends on the floating-point objects **x, y, height, width** which are directly user controlled.

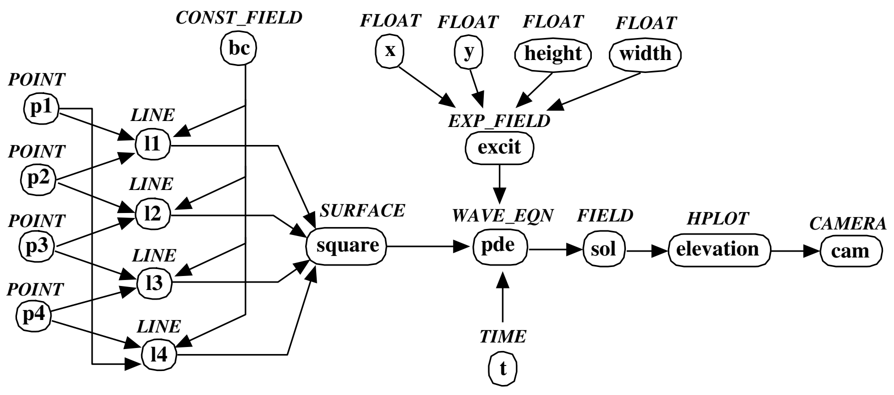


Fig. 5. Dependency graph for the wave equation simulation.

PDE: The PDE is described by an instance **pde** of the class *WAVE_EQN* which contains all specific methods for the wave PDE. This object will get automatically triggered when any of its dependencies is changed (e.g. time, excitation, geometrical domain, etc) and write a new solution to the **sol** *FIELD* object.

Time dependency: Time dependency of the PDE is modelled by a *TIME* class object **t** on which the *WAVE_EQN* object **pde** depends and which plays the role of the time node depicted in Fig. 1. The *TIME* object runs as a coroutine which advances time with a certain increment and triggers computation of a new solution.

Visualization: There is a *CAMERA* **cam** which views a *HPLOT* height plot object **elevation** of the PDE's solution.

Interactivity The user can modify the excitation directly by changing the **x, y, height, width** parameters (i.e. 'drop' a new droplet in the bucket), interact with the camera's controls, change the time step to control the simulation speed, deform the domain by moving one of the points **p1..p4**, start or stop the time **t** to pause or unpause the animation or experiment with the solver's or mesh generator parameters. Briefly, all parameters of all objects are fully accessible for modification during the simulation.

Simulating a diffusion or Navier-Stokes PDE would involve only minor modifications in the above setup (replacement of the *WAVE_EQN* object by a *DIFFUSION* or *NAVIER_STOKES* object and adjustment of boundary conditions and excitation to physically relevant values).

7 Use of the Simulation System in Engineering Problems

We used the FEM class library and the simulation system to implement a more complex numerical simulation of an electrochemical drilling (ECD) process (an electrolytic process in which the anode acts like a drill that advances into a metallic plate acting as a cathode). The anode speed and the voltage applied between the drill and the plate must be varied in time in a well controlled way in order to produce holes of a complex geometry (see Fig. 8). The problem is to find the correct voltage and speed variations in time that produce holes with the desired geometry. The simulation that we have constructed enables an engineer to vary the process parameters in real time in order to simulate the drilling of holes.

The user can control the variation of all process parameters by means of different GUI widgets like sliders and to monitor the process evolution in real time by selectively zooming in the areas of interest and/or choosing different visualization metaphors (Fig. 6 e). As parameters are changed, the system automatically performs new FE computations, domain re-meshing and presents the new solution to the user (Fig. 8). The numerical analyst can experiment by interactively changing the mesh generator or the solver objects used in the FEM simulation with different ones and monitor the convergence rate of the FE solver or change its tolerance if desired. A simple extension would be to construct an object which automatically adapts the solver's tolerance or the mesh refinement to the solution's gradient.

The ECD process is a good test case for the FEM simulation system since it essentially relies on real time user control and evolution monitoring.

8 Conclusion

Better control of complex numerical simulations of physical processes demands a general purpose simulation system which integrates modelling, computing and visualization into a single environment. This paper has presented the conceptual model and structure of a simulation system which combines the data dependency paradigm of dataflow applications with an object-oriented modelling philosophy. The result is an integrated environment in which simulations can be interactively built, steered and monitored. The system offers the power of imperative programming via an object-oriented GUI allowing visual manipulation of objects, methods and data members. Secondly, constraint programming is provided by means of an object-oriented visual specification of data dependencies.

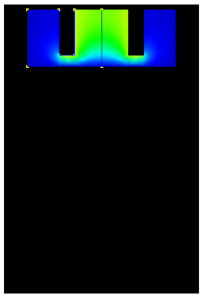
We have designed an object-oriented library for finite elements and integrated it into the presented simulation system. The combination between object-oriented numerics and interactive simulation specification yields an environment where both high level, intuitive simulation steering and fine control over numerical aspects are available. We illustrate this combination by a set of examples including a practical engineering problem.

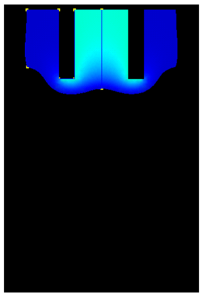
Current research goals include both the enhancement of the FEM object-oriented library with new solvers, preconditioners and support for other PDE types and investigation of better ways to interactively describe constraints in scientific simulations. A possible development considers constraint specification in terms of implicit or explicit equations or laws and an automatic conversion of these to the dependency graph representation. Another research issue regards the implementation of a full-fledged run-time C++ interpreter which should add the possibility to understand and execute complex procedural descriptions of simulations and possibly even the definition of new classes at run-time.

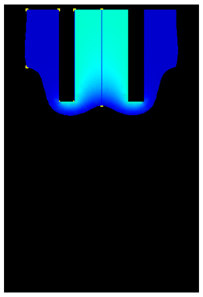
References

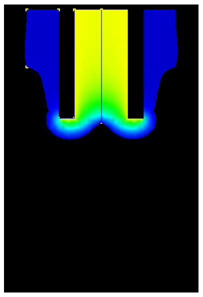
1. J. Barry, *GEOMPACK - A Software Package for the Generation of Meshes using Geometric Algorithms*, Adv. Eng. Software **13**, pp. 325–331.
2. S. Carney, M. A. Heroux, G. Li, and K. Wu, *A Revised Proposal for a Sparse BLAS Toolkit*, Army High Performance Computing Research Center Technical Report 94-034, June 1994.
3. J. Neider, T. Davis, M. Woo, *The OpenGL Programming Guide*, Addison-Wesley, 1993.
4. J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1993.
5. A. M. Bruaset, H. P. Langtangen, *A Comprehensive Set of Tools for Solving Partial Differential Equations: Diffpack*, Numerical Methods and Software Tools in Industrial Mathematics, (M. Daehlen and A.-Tveito, eds.), 1996.
6. J. J. Dongarra, R. Pozo, D. Walker, *LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra*, Proceedings of Supercomputing '93, IEEE Press, 1993, 162–171.
7. R. B. Haber, D. McNabb, *Visualization idioms: a conceptual method for visualization systems*, In *Scientific Visualization: Advances and Challenges*, Academic Press, 1994.
8. R. Marshall, J. Kempf, S. Dyer, and C. C. Yen, *Visualization methods and simulation steering for a 3D turbulence model of Lake Erie*, Computer Graphics **24**, 1990.
9. C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, *The Application Visualization System: A Computational Environment for Scientific Visualization.*, IEEE Computer Graphics and Applications, July 1989, 30–42.
10. W. Schroeder, K. Martin, B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1990
11. C. Gunn, A. Ortmann, U. Pinkall, K. Polthier, U. Schwarz, *Oorange: A Virtual Laboratory for Experimental Mathematics*, Sonderforschungsbereich 288, Technical University Berlin. URL http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html
12. B. N. Freeman-Benson, A. Borning, *Integrating Constraints with an Object-Oriented Language*, Proceedings ECOOP'92 – European Conference on Object-Oriented Programming, (O. Lehrmann Madsen, ed.), Utrecht, 1992.
13. D. H. H. Ingalls, *A Simple Technique for Handling Multiple Polymorphism*, In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pp. 347–349, November 1986.
14. R. Pozo, K. A. Remington, A. Lumsdaine, *SparseLib++: A Sparse Matrix Class Library, Reference Guide*, World Wide Web document **http://math.nist.gov/iml++/**, April 1996.

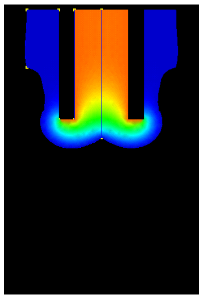
Appendix: Color Plates

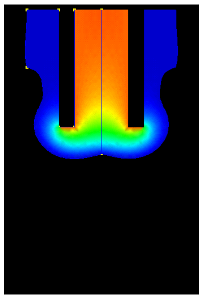

1

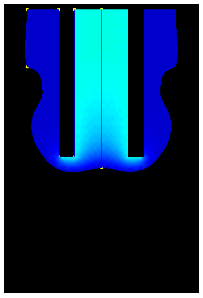

2

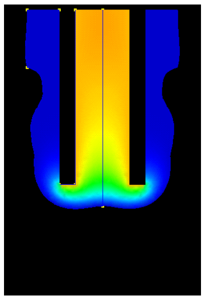

3

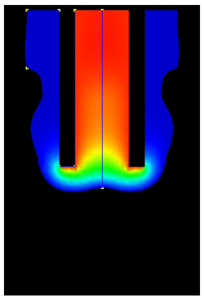

4

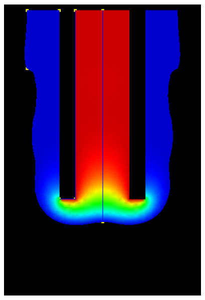

5

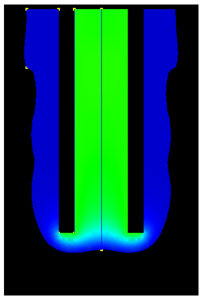

6

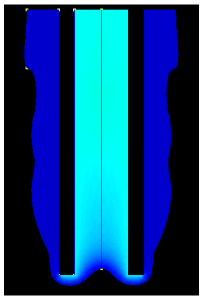

7

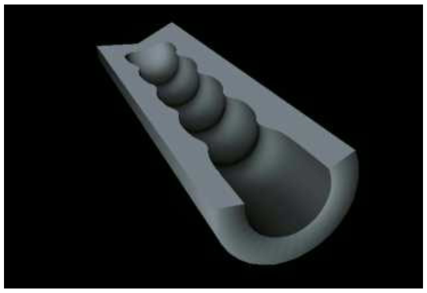

8


9


10


11


12



FEM Simulation of ECD process

1..12: electric potential at different time instants

left: section in the 3D drilled cavity

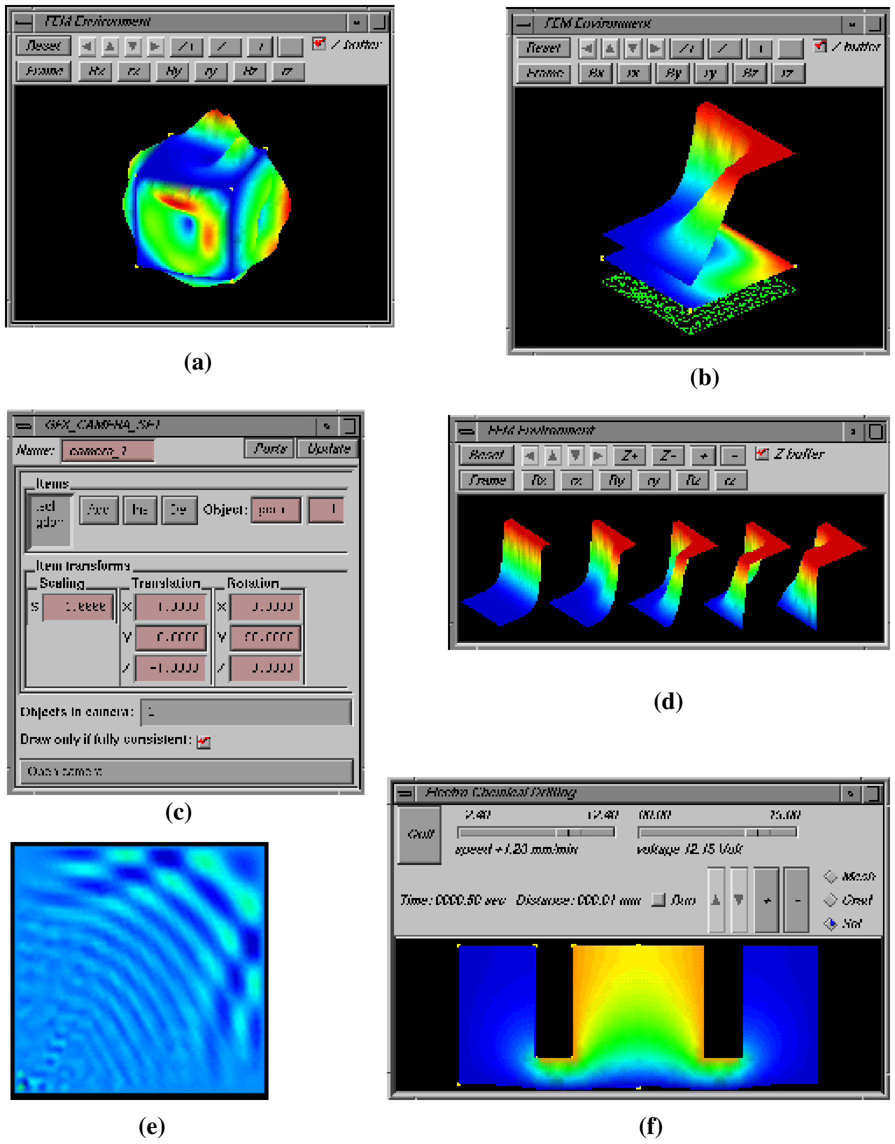


Fig. 6. Interactive FEM simulations. Diffusion process computed over a 3D cubic domain (a). Temperature solution of a free convection simulation (mesh, solution and solution 3D elevation) (b). Object-oriented interactor for the CAMERA class (c). Temperature during a time dependent free convection simulation (d). Simulation of waves (e). Interface of the ECD FEM simulation (f).