VISSION: An Object Oriented Dataflow System for Simulation and Visualization

Alexandru Telea, Jarke J. van Wijk

[1] Eindhoven University of Technology,
Den Dolech 2,Eindhoven 5600 MB, The Netherlands,
alext@win.tue.nl, http://www.win.tue.nl/math/an/alext
[2] vanwijk@win.tue.nl, http://www.win.tue.nl/cs/tt/vanwijk

Abstract. Scientific visualization and simulation specification and monitoring are sometimes addressed by object-oriented environments. Even though object orientation powerfully and elegantly models many application domains, integration of OO libraries in such systems remains a difficult task. The elegance and simplicity of object orientation is often lost in the integration phase, so combining OO and dataflow concepts is usually limited. We propose a system for visualization and simulation with a generic object-oriented way to simulation design, control and interactivity, which merges OO and dataflow modelling in a single abstraction. Advantages of the proposed system over similar tools are presented and illustrated by a comprehensive set of examples.

1 Introduction

Better insight in complex physical processes requires the combination of the visualization and interactivity (seen as the user ability to interrogate and modify the simulated universe). This has led to the advent of computational steering systems, which allow the user to change and monitor various parameters on-line and perform direct manipulation on the visualized data. To extend the user's freedom from process steering to interactive process design, the dataflow concept is often used: networks of computational modules exchanging data to perform the desired task are created by connecting module icons in a visual programming tool.

Object-oriented (OO) design is, on the other hand, the favourite technique for building extensible and reusable component libraries. Making such libraries available in a dataflow steering environment would give the end-users the conciseness, elegance and reusability of OO code, often appreciated only by code designers, but not used by the target system or lost at integration. Many environments offer steering, visualization, and code integration in various amounts, but no single one addresses these and the extra requirement of existing OO libraries integration in a unitary, easy to learn manner.

We addressed the above problem by designing VISSION, a general purpose environment for VIsualization and Steering of SImulations with Objectual Networks. Dataflow modelling familiar to visualization scientists [4, 6] is completely merged

with the OO modelling used by component designers [5, 2] in a single new abstraction. Independently developed OO code integration is thus almost transparent, especially since VISSION automatically constructs its GUIs from the given code, extending the approach presented in [7]. This paper presents VISSION from a user perspective, its object-oriented design being detailed in [9].

This paper is organized as follows: Section 2 presents the main requirements of generic simulation systems and the main limitations of existing systems. Section 3 shows how VISSION fulfills these requirements. Applications of VISSION are presented in Section 4. We conclude the paper presenting further research directions.

2 Background

In the most demanding scenario, an open environment targets three user categories: end-users (EU) steer a simulation via virtual cameras, direct manipulation, GUI widgets, or interpreted command languages. Application designers (AD) build applications for a wide range of EU domains and thus require simple to use, yet generic interactive tools to select and assemble domain-specific components [15]. Component developers (CD) build these components and require that existing code should be easily extensible and reusable as modular components, and that the target environment should not constrain their design. Often the same person goes through all three roles (e.g. a researcher who develops his own code as a CD, then builds experiments to test algorithms as an AD, and finally monitors and/or steers the final application as an EU). The cycle repeats, (EU insight triggers application design revisions, which may ask for new/specialized components), so the role transition should be transparent: CD's code should be immediately available to the AD, who should easily produce the EU's end-application.

There is hardly any visualization/simulation system which fulfills the above requirements union and offers a simple, yet generic solution for the role transition. Turnkey systems (Fig. 1) have custom tools and GUIs to excel in specific tasks, are easy to learn and use, but are by definition not extensible or customizable. OO libraries [2, 5, 3] are highly customizable and extensible, but require manual programming of data flows and GUIs. Dataflow systems [4] are

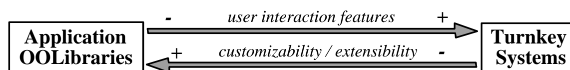

Fig. 1. A flexible system should combine the customizability/extensibility of OO libraries with the usability of the turnkey systems.

extensible and customizable, as simulations are interactively built by connecting user-written modules, but still have limitations. Few support both by-value and

by-reference data transfer between modules (limitation *L1*), even fewer support user-defined types for the modules' inputs and outputs (limitation *L2*). Many such systems use different languages for module implementation, user interface, scripting, and dataflow control, making them hard to learn and use (limitation *L3*). As there is often no way to map constructs from an (OO) language to another, developers are forced to use the languages' common subset(limitation *L4*) [6, 5, 4], or to manually adapt their code (limitation *L5*). The set of system GUI widgets is usually not extensible to reflect directly e.g. user-defined types (limitation *L6*). *L5* and *L6* imply that GUI construction can not be automated (limitation *L7*). Few systems allow writing new modules by reusing and/or combining the existing ones and programming only the new features (no support for module inheritance, limitation *L8*). Some systems enhance monolithic simulations with dataflow-based tracking / steering features by manual insertion of data transfer and synchronization code [16, 11, 10, 14]. These systems provide no inherent support for the CD, as they have no 'component' notion.

3 Overview of the System

The integration of dataflow/visual programming with component OO modelling comes naturally as the presented requirements are fulfilled complementarily by dataflow systems (interactivity,visual programming,GUI construction,steering) and OO application libraries (customizability, extensibility,high-level modelling), Some systems [4],[6] take this path, but none *combine dataflow modelling and OO modelling in a single abstraction*, so the listed problems are merely alleviated. VISSION completely merges OO and dataflow modelling in a new abstraction called a *metaclass*, used as its fundamental concept. The following shows how this addresses the outlined limitations and requirements.

3.1 The Metaclass Concept

From the OO modelling viewpoint, modules are implemented as C++ classes, organized by the CD as various application domain *libraries*. From the dataflow viewpoint, a module (called a *metaclass*) is an entity which enhances a C++ class with a *dataflow interface*, i.e. a set of typed input and output *ports* and an update procedure. The ports and update procedure are specified in terms of the C++ class's public methods and members: when a port is read/written, a C++ member's value is read/written or a method is called and the return value is used. Ports are typed by the C++ types of their underlying class members. Metaclasses are object-oriented entities, so they can inherit from each other, thus enabling the reuse of existing metaclasses to create new ones (addresses limitation *L8*). All information needed to 'promote' a C++ class to be directly loadable by VISSION resides in its metaclass. Our solution differs fundamentally from other systems asking the user to 'insert' system calls in his code to make it available to the framework [4],[10],[11] or to inherit from a system base class [2], [5] and hence addresses limitation *L5*.

Metaclasses:	*C++ classes:*

node **IVSoLight**
{ input:
WRPort "intensity" (setIntensity,getIntensity)
editor: Slider
WRport "color" (setColor,getColor)
WRport "light on" (on)
}
	class **IVSoLight**
	{ public:
	BOOL on;
	void setIntensity(float);
	float getIntensity();
	void setColor(IVSbColor&);
	IVSbColor getColor();
	};

node **IVSoDirectionalLight**: **IVSoLight**
{ input:
WRPort "direction" (setDirection,getDirection)
}
	class **IVSoDirectionalLight:** public **IVSoLight**
	{ public:
	void setDirection(IVSbVec3f&);
	IVSbVec3f getDirection();
	};

Fig. 2. Example of C++ class hierarchy and corresponding metaclass hierarchy

Figure 2 exemplifies the above for two C++ classes and their metaclasses: the IVSoLight metaclass has three inputs for a light's color, intensity, and on/off value, implemented by the corresponding class's methods with similar names, and of types IVSbColor (a RGB triplet), float, and respectively BOOL. IVSoDirectionalLight extends IVSoLight with the light's direction, of type IVSbVec3f (a 3D vector). The user can easily specify other information in the metaclass, such as GUI and widget preferences, and help data. The appropriate widgets are automatically constructed based on the ports' types (3 float typeins for the vector and the RGB color, a toggle for the boolean, and a slider, as the preference specified, for the float). Separating this information from the C++ class lets us enhance existing classes with dataflow/GUI features non-intrusively. It has also let us develop a generic persistence scheme to save a simulation as a C++ source file. This addresses limitation $L4$, as no custom file format was needed (see [8] for a similar approach).

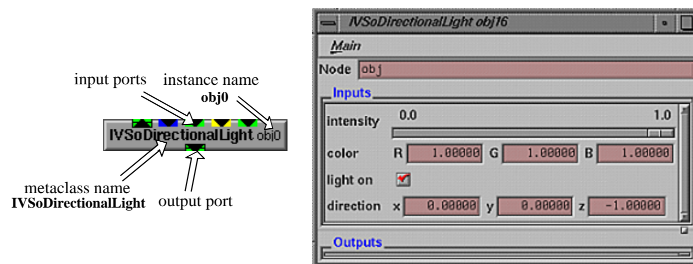

Fig. 3. Left: Visual representation of a metaclass. The various graphical signs used for the ports encode port C++ type, by value/by reference transfer, and other attributes. Right: automatically constructed GUI for the metaclass

3.2 Features

VISSION allows the user to load the desired metaclass libraries, browse a palette with the loaded metaclasses, create new nodes (i.e. instances of metaclasses), connect, clone, or delete existing nodes in a GUI similar to [4, 6] (Fig. 4). The fundamental differences between VISSION and similar systems appear as we look at the dataflow semantics.

The Dataflow Mechanism The dataflow mechanism is entirely based on the object-oriented typing offered by the C++ language, that is, data can be passed by value, by pointers or by reference, and can be of any type (addresses limitations *L1* and *L2*). If class types are used, constructors and destructors are properly invoked when data elements flow from the output to the input port. Secondly, connections between ports obey the full OO typing of by C++: a port of type A can be connected to a port of type B if the C++ type A conforms to the C++ type B. All C++ type conversions [1] are used: trivial conversion, subclass to baseclass, constructors, and conversion operators. The user interactively builds networks using the same types, rules and checking he would use in a C++ compiled program. We believe the above is a sound generalization of the dataflow typing used by other systems: The Oorange system, based on *Objective C*, offers by-reference but no by-value transfer. AVS/Express limits data types to its own *V language* which is far less powerful than C++ (e.g. it lacks constructors, destructors and multiple inheritance). Compiled libraries (e.g. vtk) are only statically extensible, as all types have to be known at compile time, which makes them unsuitable for a dynamic, interactive modelling environment.

To cope with complex networks, VISSION offers *node groups* containing subgraphs up to an arbitrary depth, which can be interactively constructed by adding nodes and ports to an empty group. This generalizes Oorange's nodes, AVS's macros, and Inventor's node kits. A less common feature is the support for networks having several *loops*, which allows a very natural way to describe iterative processes, or to implement direct manipulation as dataflows that go 'upstream' from the camera modules to other 'data processing' modules. We make no distinction between up and down stream (as compared to [4]), as the network traversal copes with any directed cyclic graph.

The GUI Interactors GUI interaction panels (shortly interactors) are provided to examine and modify the values of the nodes' ports, connect or disconnect ports, or perform actions on nodes. Interactors create the third object hierarchy in the system, isomorphic with the C++ class and metaclass hierarchies. The widgets of an interactor are based on the types of their ports: a *float* port can be edited by a slider, a *char** port by a textual type-in, a three-dimensional *VECTOR* port by a 3D widget manipulating a vector icon in 3-space, a *boolean* by a toggle button, etc (Fig. 5, 3). The set of GUI widgets for the basic types can be extended by the AD with widgets for user-defined types. This allowed us to provide GUI widgets for some types of specific libraries, such as 3D vectors, colors, rotation matrices, light values, etc. This addresses limitation *L6*.

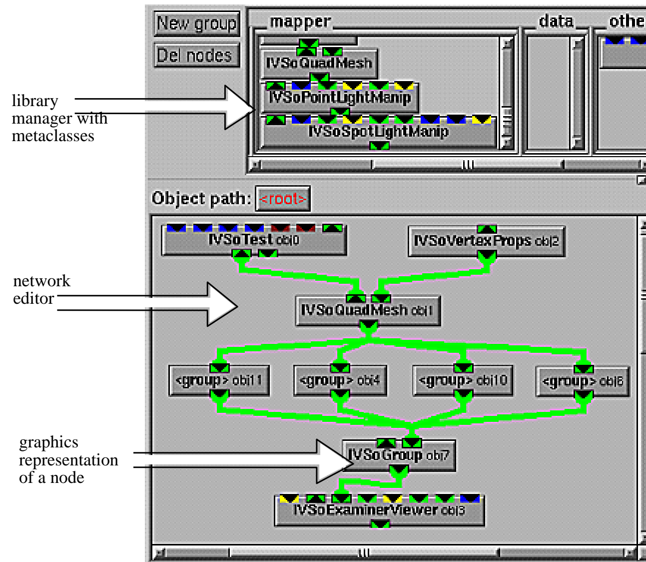

Fig. 4. The network editor and the dataflow graph. Nodes are created from metaclasses shown in the library manager's GUI.

VISSION automatically associates widgets with port types by picking out of the basic/custom widgets the one whose type best matches the port's type. The AD can thus customize the look of an application GUI, either by creating new GUI widgets or by associating the existing ones with other types (e.g. prefer a float type-in instead of a slider for a float port), and still have the interactors built automatically (this addresses limitation *L7*). Finally, the EU can instantly change a port's widget at run-time via a menu listing all widgets capable to edit that port. Dynamically associating OO widgets to OO types enables thus the creation of *user interface libraries* that can be transparently reused by VISSION to steer any network reading/writing compatible types.

The system offers also a GUI for direct inspection and modification of the C++ objects used by the metaclasses, similar to the visual class browsers of several OO compilers or debuggers. This allows CDs to directly test their C++ classes bypassing the metaclass abstraction level. Finally, the EU can type commands directly in C++ in a GUI window to be interpreted (Fig. 5), an interaction mode preferred by some users over the widget metaphor, or load and execute C++ source code. This allows the EU to write animations based on arbitrarily complex control sequences directly in C++ without having to learn a new animation language (see Fig 7c for an example of a finite element simulation based animation). Limitation *L4* is addressed, as C++ is VISSION's single language used for application class coding, dataflow typing, run-time command-line user interaction, and persistence.

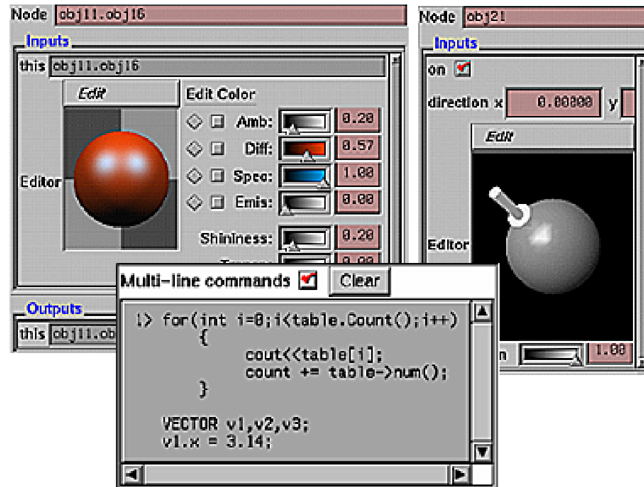

Fig. 5. GUI widgets for interaction with metaclasses

3.3 Implementation

VISSION consists of three main parts: the object manager, the dataflow manager, and the interaction manager (Fig. 6), based on two lower level components: the C++ interpreter and the library manager, communicating by sharing the dataflow graph. The key element (which enabled us to elegantly and easily remove the limitations exhibited by similar systems) is a *C++ interpreter*. Port connections/disconnections, data transfer between ports, invocation of node update methods, GUI-based inspection and modification of ports, automatic GUI construction, and interpreted scripts are uniformly implemented as small C++ fragments dynamically sent to the interpreter. The interpreter cooperates with the *library manager* to dynamically load application libraries containing metaclass declarations and their compiled C++ classes, with the *object manager* to create and destroy the metaclasses, and with the *interaction manager* to build and control the GUIs. All application code is executed from compiled classes, leaving a very small C++ code amount to be interpreted. The performance loss as compared to a 100% compiled system was estimated to be lower than 2%, even for complex networks intensively accessing the interpreter.

4 Applications

The following presents some of the applications we have built with VISSION.

Scientific Visualization We have chosen the Visualization Toolkit (shortly vtk) [5], one of the most powerful freely available scientific visualization libraries,

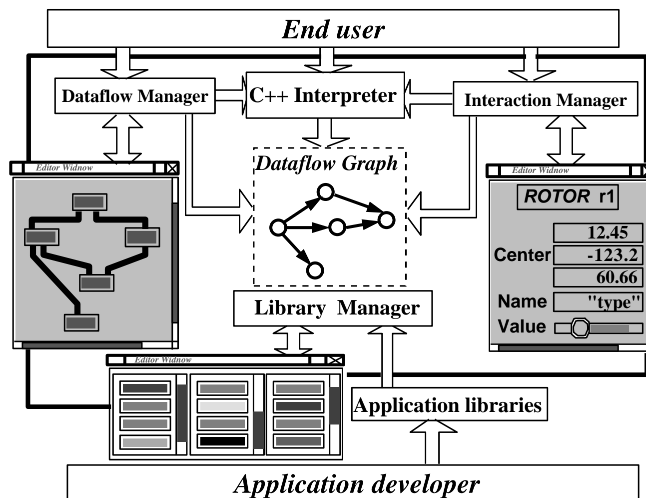

Fig. 6. Architecture of the simulation and visualization system

and integrated it into VISSION. Since vtk is a C++ library, its integration didn't pose any problems (keeping the constraint of not modifying its source code).

As a rendering back-end we used Open Inventor, which we also fully integrated. The EU can pick any vtk or Inventor class (of the total of approximately 250, respectively 70) in the visual browser, instantiate it, and connect it with other nodes, without knowing C++ or even knowing they are written in C++. We had to write a single 'adapter-like' class of around 120 C++ lines to connect all the Inventor rendering and direct manipulation facilities (superior to vtk's rendering classes, which we didn't use) to the vtk pipeline.

Scalar, vector, tensor, and medical visualizations were created with the vtk-Inventor metaclasses (Fig. 7 a,i,g, Fig 7 f, Fig 7 e, respectively Fig 7 j) with practically the same ease as if using AVS or other similar system. The integration required writing around 320 metaclasses, of an average length of 6 text lines, and absolutely no change to the two libraries (of which, Inventor was not even available as source code).

Global Illumination Radiosity simulation software often requires delicate tuning of many input parameters, and thus can not be used as black box pipelines. Testing new algorithms requires also the configurability of the radiosity pipeline. These options are however rarely available to non-programming experts in current radiosity software. We addressed this by including a progressive refinement radiosity system written in C/C++ by us before VISSION was conceived, into VISSION. The 3D world tessellation with vertex intensities output was easily made available for visualization in the Inventor library by the creation of an 'adapter' module. Users can now change all the 'hidden' parameters along the radiosity

pipeline, easily insert new algorithms for e.g. sharp shadow detection [12] by subclassing, and visually monitor the process convergence (Fig. 7 d).

Finite Element Simulations Finite element (FE) applications mostly come as packages that limit the user's interaction input given as batch files, respectively an output visualized in a post simulation phase. We addressed these limitations by integrating our FE C++ library [7] in VISSION. Researchers can specify and solve FE problems interactively, experiment with different numerical techniques, and monitor error and convergence rates, without quitting the environment to redefine input files or recompile. Examples include 3D diffusion problems (Fig. 7 a), time-dependent free convection problems (Fig. 7 c), wave simulations (Fig. 7 h), or industrial steering turn-key software [13]. (Fig. 7 b). Visualization is performed again by the Inventor library.

5 Conclusion

We have presented VISSION, a general-purpose visualization and computational steering system built on an object-oriented foundation. VISSION is a generic environment the specification, monitoring, and steering of simulations which removes some limitations of similar systems by combining the powerful, yet so far independently used OO and dataflow modelling concepts.

We have enhanced the traditional dataflow mechanism used by simulation systems to an object-oriented one by introducing the metaclass concept, which extends C++ classes with dataflow semantics in a non intrusive manner. Adding application code to the system is greatly simplified as compared to similar systems. Application library design is clearly separated from the system-specific dataflow information held in the metaclasses. We have provided a mechanism for automatic GUI construction for application modules based on the OO metaclass semantics, and a way to add type-specific, user-defined widgets, based on OO typing.

Several applications illustrate the advantages of a fully object-oriented and single language architecture. Component designers have included libraries for scientific visualization and rendering (420 classes), radiosity (18 classes) and finite element analysis (25 classes) in the system in a short time (approximately 2 months, 5 days, 10 days respectively), while application designers and end users could effectively use the system in a matter of minutes.

Future work is aimed at the extension of VISSION's OO aspects with features such as class hierarchy browsing, automatic documentation, and a generalization of the dataflow model to include also *code flow*, that is to have modules synthesize, exchange, and execute C++ code fragments, creating multiple new possibilities for modelling simulations. Parallel work targets the inclusion of other application domains as numerical iterative solvers or computer vision interfaces and their coupling with the already available libraries.

References

1. B. STROUSTRUP, *The C++ Programming Manual*, Addison-Wesley,1993.
2. J. WERNECKE, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1993.
3. 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.
4. 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.
5. W. SCHROEDER, K. MARTIN, B. LORENSEN, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1995
6. 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
7. A.C. TELEA, C.W.A.M. VAN OVERVELD, *An Object-Oriented Interactive System for Scientific Simulations: Design and Applications*, in /textitMathematical Visualization, H.-C. Hege and K. Polthier (eds.), Springer Verlag 1998
8. B. MEYER, *Object-oriented software construction*, Prentice Hall, 1997
9. A. C. TELEA *Design of an Object-Oriented Computational Steering System*, in *Proceedings of the 8th ECOOP Workshop for PhD Students in Object-Oriented Systems*, ECOOP Brussels 1998, to be published
10. J. J. VAN WIJK AND R. VAN LIERE, *An environment for computational steering*, in G. M. Nielson, H. Mueller and H. Hagen, eds, *Scientific Visualization: Overviews, Methodologies and Techniques*, computer Society Press, 1997
11. S. RATHMAYER AND M. LENKE, *A tool for on-line visualization and interactive steering of parallel hpc applications*, in *Proceedings of the 11th International Parallel Processing Symposium*, IPPS 97, 1997
12. A.C. TELEA AND C. W. A. M. VAN OVERVELD, *The Close Objects Buffer: A Sharp Shadow Detection Technique for Radiosity Methods*, the *Journal of Graphics Tools*, Volume 2, No 2, 1997
13. M. J. NOOT, A. C. TELEA, J. K. M. JANSEN, R. M. M. MATTHEIJ, *Real Time Numerical Simulation and Visualization of Electrochemical Drilling*, in *Computing and Visualization in Science*, No 1, 1998
14. D. JABLONOWSKI, J. D. BRUNER, B. BLISS, AND R. B. HABER, *VASE: The visualization and application steering environment*, in *Proceedings of Supercomputing '93*, pages 560-569, 1993
15. W. RIBARSKY, B. BROWN, T. MYERSON, R. FELDMANN, S. SMITH, AND L. TREINISH, *Object-oriented, dataflow visualization systems - a paradigm shift?*, in *Scientific Visualization: Advances and Challenges*, Academic Press (1994), pp. 251-263.
16. S. G. PARKER, D. M. WEINSTEIN, C. R. JOHNSON, *The SCIRun computational steering software system*, in E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 1-40, Birkhaeuser Verlag AG, Switzerland, 1997

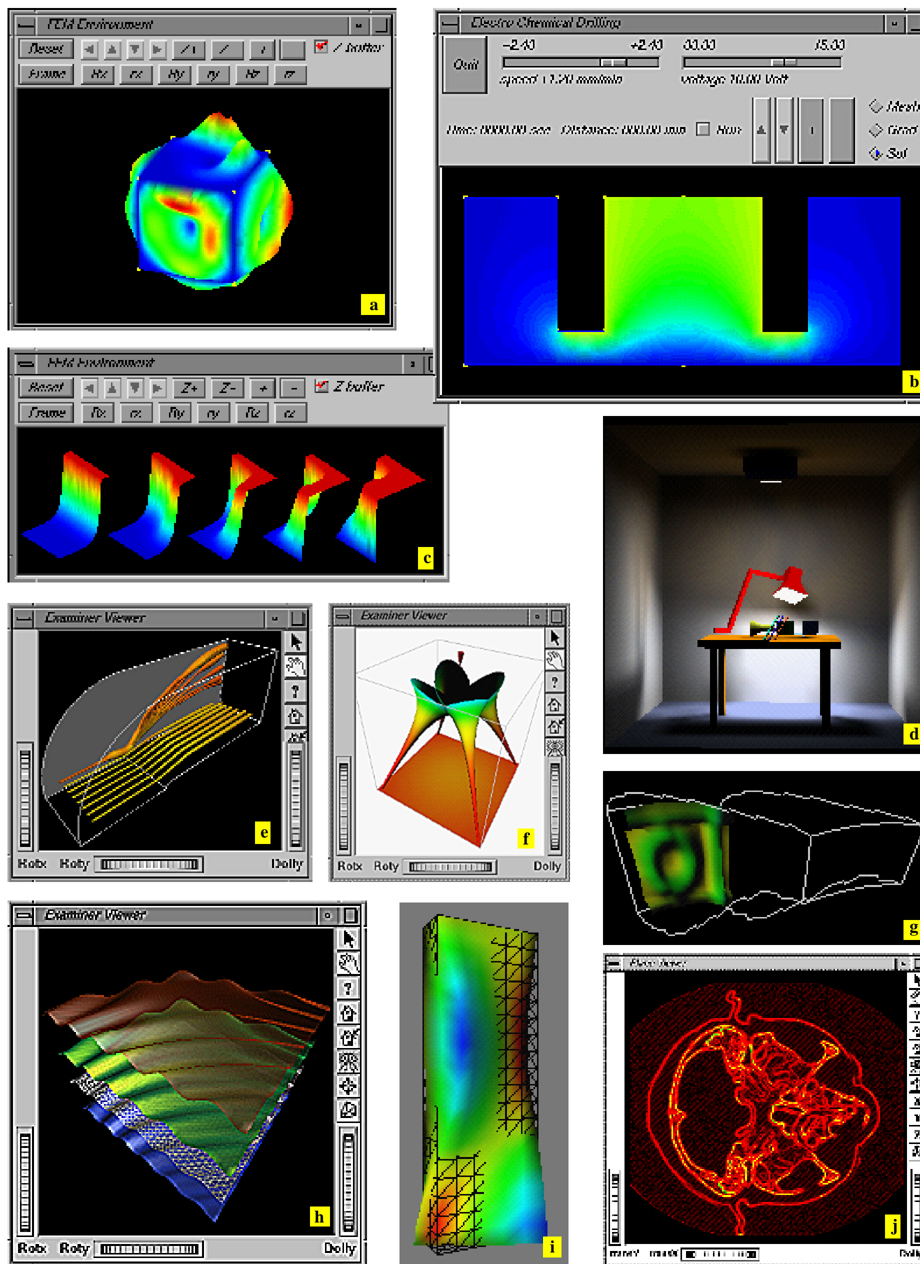

Fig. 7. Visualizations and simulations performed in the VISSION environment