Simulation and Visualization in the VISSION Object Oriented Dataflow System

A.C. Telea J.J. van Wijk

Department of Mathematics and Computing Science
Eindhoven University of Technology, The Netherlands
alext@win.tue.nl vanwijk@win.tue.nl

Keywords: scientific visualization, steering, dataflow, object-oriented systems

Abstract

Scientific visualization and simulation steering and design are mostly addressed by non object-oriented (OO) frameworks. Even though OO powerfully and elegantly models many application areas, integration of OO libraries in such systems remains complex. The power and conciseness of object orientation is often lost at integration, as combining OO and dataflow concepts is very limited. We propose a visualization and simulation system with a generic object-oriented way to simulation design, control and interactivity, merging OO and dataflow modelling in a single abstraction. Advantages of the presented system over similar ones are illustrated by a comprehensive set of examples.

1 Introduction

Better insight in complex simulations has led to the advent of computational steering systems, which change, monitor the simulation parameters and directly manipulate the visualized data. Dataflow systems add to this interactive process design: networks of computational modules exchanging data to perform the desired task are created by connecting icons in a visual programming tool. Object-oriented (OO) design is, on the other hand, the favourite approach to building extensible and reusable component libraries. Making the components of such libraries available to the interactive process design in dataflow steering systems would give the end-users the conciseness, elegance and reusability of OO code, often appreciated only by code designers. Many systems offer steering, visualization, and code integration, but no single one addresses all these and the extra requirement of interactive integration and component manipulation from existing OO libraries in a unitary, easy to learn way.

We addressed the above by designing VISSION, a generic environment for VIsualization and Steering of SImulations with Objectual Networks. VISSION is founded on a new abstraction which fully merges the dataflow concept [2, 4] with OO modelling [3, 1, 12]. Existing/new OO code integration is almost transparent, especially since VISSION automatically constructs its GUIs from the given code (extending the approach presented in [5]). This paper presents VISSION from a user perspective, its OO design is detailed in [6]. Section 2 lists the main requirements of generic simulation systems and the main limitations of existing ones. Section 3 shows how VISSION fulfills these requirements. Applications of VISSION are presented in Section 4. We conclude the paper with further research directions.

2 Background

We target the needs of three user groups: end-users (EU) need to steer simulations via virtual cameras, direct manipulation, and GUI widgets. Application designers (AD) build applications for various EU domains and thus need generic (interactive) tools to select and assemble domain-specific components [10]. Component developers (CD) need to build/extend/reuse the components easily (not constrained by the target environment). Often the same person takes all three roles (e.g. a researcher who writes own code as a CD, builds experiments as an AD, and then monitors / steers the final application as an EU). The cycle repeats, (EU insight triggers application redesign, which may ask for new/specialized components), so role switching should be easy. Hardly any visualization/simulation system covers all these demands and has a simple, yet generic solution for the role transition. Turnkey systems excel in custom tools/GUIs for specific tasks, are easy to learn and use, but are by definition not extensible. OO libraries [1, 3, 12] are highly extensible, but need manual programming of data flows and GUIs. Dataflow systems using visual programming

[2] are extensible and customizable, but still have limitations. Few support both by-value and by-reference data transfer between modules (limitation *L1*), even fewer allow user-defined types for the modules' inputs and outputs (limitation *L2*). Many systems use different languages for module implementation, user interface, scripting, and dataflow control, making them hard to learn and use (limitation *L3*). Constructs from an (OO) language often don't map to another, so CDs must use the languages' common subset(limitation *L4*) [4, 3, 2], or manually adapt their code (limitation *L5*). The set of system GUI widgets is often not extensible (e.g. to edit directly new data types) (limitation *L6*). *L5* and *L6* imply that GUI construction can not be automated (limitation *L7*). Few systems support module inheritance, i.e. creating new modules by reusing and/or combining existing ones and writing only the new features (limitation *L8*). Another class of applications such as tracking and steering systems enhance monolithic simulations by manual insertion of 'system calls' [11, 8, 7] and thus provide no support for the CD, as they have no 'component' notion.

Overall, simulation and visualization software can be seen as spanning a continuum between two extremes (see Fig. 1). At one end of the spectrum, (object-oriented) application libraries offer the 'pure' computational code blocks with full control on customization and extensibility by using the programming language mechanisms of their design language. Libraries are however not directly usable by end users, as they have to be manually provided with e.g. GUIs and control flow in an end-application context. At the other extreme, turnkey systems are directly (and easily) usable by end-users via specialized GUIs and other interaction policies. Their extensibility and customizability is however quite limited, as they are usually designed to cover specific, fixed application domains. The challenge we see is to provide a system which covers the entire continuum, offering various mechanisms (extensibility, customizability, interactive control) for various user classes (component designers, application designers, end users respectively).

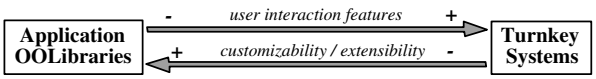

Figure 1: Application systems trade-offs

3 Overview of VISSION

Integration of dataflow/visual programming with component OO modelling comes naturally as all previous demands are fulfilled complementarily by dataflow systems (interactivity,visual programming,GUI construction,steering) and OO libraries (customizability, extensibility,high-level modelling). Some systems [2],[4] take this path, but none *merges dataflow and OO concepts in a single concept*. The listed problems are merely alleviated. VISSION's fundamental concept, called *metaclass*, completely merges OO and dataflow modelling, and addresses the presented demands as follows.

From the OO modelling viewpoint, modules are implemented as C++ classes, organized by the CD in application *libraries*. From a dataflow perspective, a module (called a *metaclass*) enhances a C++ class with a *dataflow interface*, i.e. a set of typed input and output *ports* and an update function. The ports and update function are specified in terms of the C++ class's public part: when a port is read/written, a C++ method is called to read/write the port's data. Ports are typed by the C++ types of their underlying 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*). Besides ports, metaclasses can hold help data and GUI preferences. All information used to 'promote' a C++ class to be loadable by VISSION resides in its metaclass. Our solution differs fundamentally from most systems forcing users to insert system calls in their code [2, 7, 8] or to inherit from a system base class [1], [3] and thus addresses limitation *L5*.

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 state, implemented by its C++ class's methods with similar names, and of types IVSbColor (a RGB triplet), float, and respectively BOOL. IVSoDirectionalLight adds to IVSoLight the light's direction, of type IVSbVec3f (a 3D vector). Suitable widgets are automatically constructed from 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 (addresses *L4*). In VISSION the user can 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 [2, 4] (Fig. 4). The main differences between VISSION and these systems appear as we look at the dataflow mechanism.

3.1 The Dataflow Mechanism

Our dataflow mechanism is based on the full typing offered by C++, implemented *dynamically*: data can be passed between modules by value, by pointers or by reference, and can be of any type (addresses limitations *L1* and *L2*). The data passing facilities we offer are practically limited only by language mechanisms,

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();
	};

Figure 2: C++ class hierarchy and corresponding metaclass hierarchy

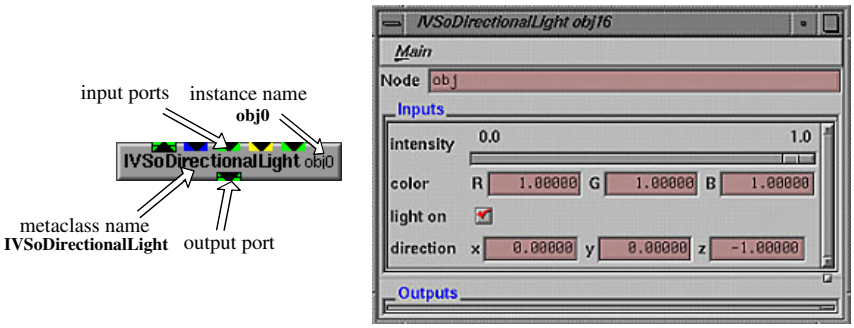

Figure 3: Left: Visual icon for a metaclass. Several graphical signs encode the ports' C++ types, by value/by reference transfer, and other attributes. Right: automatically built GUI for the metaclass

not by system mechanisms. This offers not only much greater flexibility as compared to other systems which implement a limited set of data passing facilities, but also a formal comprehensive framework for data passing, as described by C++'s language mechanisms.

A port of type *A* can thus connect to one of type *B* if the C++ type *A* conforms to the C++ type *B* by trivial conversion, subclass to baseclass, constructors, and conversion operators. The user interactively builds networks having the same type checking and freedom he would enjoy in a C++ compiled program. This powerfully generalizes the dataflow typing used by other systems: Oorange, 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++ (it lacks constructors, destructors and multiple inheritance). Compiled tools (vtk, Inventor) are only statically extendable, as all types must be known at compile time, making them unsuitable for a dynamic, interactive modelling environment.

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. We generalize Oorange's nodes, AVS's macros, and Inventor's node kits by offering the possibility to create *group types*, i.e. 'subnetwork templates' which can be saved in libraries and instantiated later, exactly as modules can. From the application designer's perspective, group types and metaclasses are identical, i.e. they are manipulated identically in the network editor and the library browser. The difference between group types and metaclasses appears at the component designer level: while metaclasses require writing C++ code and the metaclass specification (and thus some knowledge of C++ / metaclass design and language syntax), group types can be designed *interactively* and thus offer a programming-free manner to create new types by combining existing ones.

Finally, VISSION supports networks having *loops* with no distinction between up and down stream directions(as compared to [2]). This is a natural way to describe iterative processes or to implement direct manipulation as dataflows from camera modules to other data processing modules.

VISSION consists of three main parts: the object manager, the dataflow manager, and the interaction manager (Fig. 5), based on two lower level components: the C++ interpreter and the library manager, communicating by sharing the dataflow graph. The element enabling us to elegantly and easily remove the limitations similar systems have is a *C++ interpreter* [13] which constitutes VISSION's kernel element. Port connections/disconnections, data transfer between ports,update method invocation, GUI-based

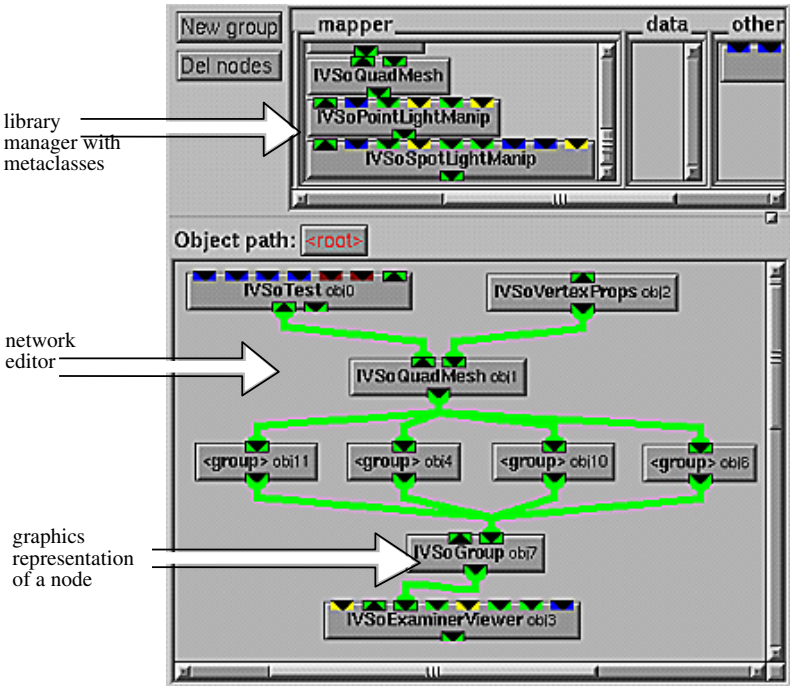

Figure 4: The network editor and the dataflow graph. Nodes are created from metaclasses shown in the library manager's GUI.

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 metaclasses and their compiled C++ classes, with the *object manager* to create and destroy nodes, 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. We estimated the performance loss as compared to a 100% compiled system to be under 2%, even for complex networks.

3.2 The GUI Interactors

GUI interaction panels (shortly interactors) are provided to examine and modify the nodes' ports, connect or disconnect ports, or perform actions on nodes. Interactors create the third object hierarchy in VISSION, isomorphic with the C++ class and metaclass hierarchies. The widgets of an interactor are based on their ports' types: 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. 6, 3). The set of GUI widgets for 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*. VISSION automatically associates widgets with port types by choosing from the given widgets the one whose type best matches the port's type. The best match rules we designed for selecting a widget to edit a given type are based on a distance metric in type space: if a widget specially designed for a given type exists, it will be used to edit that type, else VISSION selects a widget whose type can be converted to the type to edit by a minimal number of conversions, similarly to the conversion rules of the C++ language. If no custom widget is available for a given type, VISSION uses a generic widget that offers a text-based editor for any C++ type.

The AD can thus customize the look of a 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*). Note in this sense that widgets can be designed after and independently of the types they edit, An application's GUI can thus be enriched with a better GUI even after all its components have been designed, and even at run-time, by dynamically loading a new widget library. After the new widgets are loaded, the EU can customize the GUIs of his running application simply by clicking on its widgets and selecting other compatible widgets to be used instead.

Finally, the EU can type commands directly in C++ in a GUI window to be interpreted (Fig. 6), 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). Remark that the choice for C++ as a scripting and command-line language imposes no higher learning burden on the average end user as compared to systems using simpler run-time languages. Although using C++ for designing complex applications and frameworks is more complex and has a slower learning curve than e.g. using C or Fortran, our case is different: using C++ *as a scripting language* means that only its simplest features will actually be employed (e.g. control structures such as loops and tests), which are no more complex to write than the ones offered by any scripting language. However, advanced users can directly employ the more sophisticated language features of C++ if they desire. Limitation *L4* is thus addressed, as C++ is VISSION's single language for application class coding, dataflow typing, and run-time command-line interaction.

4 Applications

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

4.1 Scientific Visualization

We chose the Visualization Toolkit (vtk) [3], one of the most powerful freely available scientific visualization libraries, and integrated it into VISSION. As a rendering back-end we fully integrated the Open Inventor library. The EU can interactively pick any vtk or Inventor class, instantiate it, and connect it with other nodes, without knowing C++ or even knowing they are written in C++. We wrote a single 'adapter' class of around 120 C++ lines to connect all of Inventor's rendering and direct manipulation (superior to vtk's rendering, which we didn't use) to the vtk pipeline. Scalar, vector, tensor, and imaging visualizations were created with the vtk-Inventor metaclasses (Fig. 7 a,i,g, Fig 7 f, Fig 7 e, respectively Fig 7 j) with the same ease as if using AVS or a similar system. The integration of both libraries as visual components 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).

4.2 Global Illumination

Radiosity simulations often requiredelicate 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. Non-programming experts however rarely have these options in current radiosity software. We addressed this by including a radiosity system written in C/C++ by us before VISSION was conceived, into VISSION. Its output (3D mesh with vertex intensities) was easily passed for visualization to Inventor by the creation of an 'adapter' module. Users can now change all the 'hidden' parameters along the radiosity pipeline, easily insert new algorithms by subclassing (e.g. for sharp shadow detection [14]), and visually monitor the process convergence (Fig. 7 d).

4.3 Finite Element Simulations

Finite element (FE) software mostly comes as packages providing interaction by a batch file input/output. We addressed this limitation by integrating our FE C++ library [5] in VISSION. Researchers can now interactively model and solve FE problems, experiment with different numerical techniques, and monitor error and convergence rates. 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 [9]. (Fig. 7 b). Visualization is performed again by the Inventor library.

5 Conclusion

VISSION is a generic environment for simulation specification, monitoring, and steering which removes many 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 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 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, modular GUI construction based on the OO metaclass concept, and a way to add type-specific, user-defined widgets, based on OO typing. Component designers included libraries for scientific visualization and rendering (420 classes), radiosity (18 classes) and finite elements (25 classes) in VISSION in a short time (approximately 2 months, 5 days, 3 days respectively). Application designers and end users could use VISSION in a matter of minutes. We aim to extend 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 modeling simulations. While the implementation of code flow is almost trivial in VISSION due to its C++ interpreter engine capability to parse and execute C++

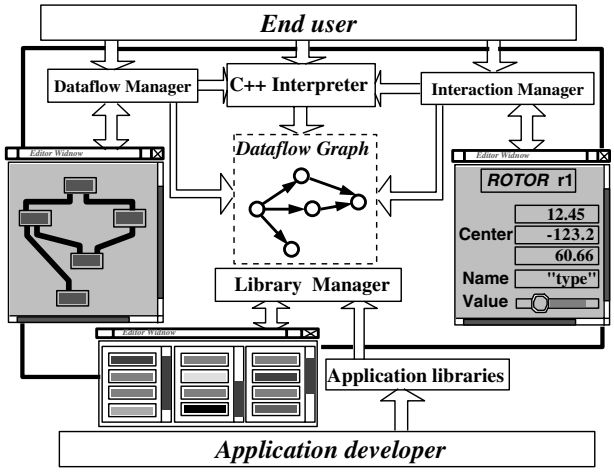

Figure 5: Architecture of the simulation and visualization system

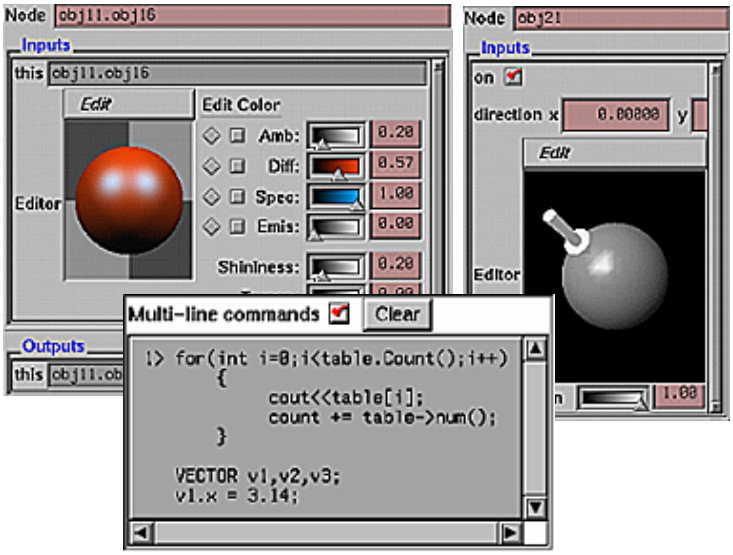

Figure 6: GUI widgets for interaction with metaclasses

code fragments dynamically, a formal framework of the code flow paradigm still needs to be developed. Besides this, we try to include other application areas as computer vision interfaces and feature tracking. More references on VISSION ar available online, at http://www.win.tue.nl/math/an/alext.

References

[1] J. WERNECKE, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1993.

[2] 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.

[3] W. SCHROEDER, K. MARTIN, B. LORENSEN, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1995

[4] 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

[5] A.C. TELEA, C.W.A.M. VAN OVERVELD, *An Object-Oriented Interactive System for Scientific Simulations: Design and Applications*, in /textit-Mathematical Visualization, H.-C. Hege and K. Polthier (eds.), Springer Verlag 1998

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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.

[11] 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

[12] 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.

[13] M. GOTO, *The CINT C/C++ Interpreter and Dictionary Generator*, The ROOT System URL http://root.cern.ch/root/Cint.html

[14] 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

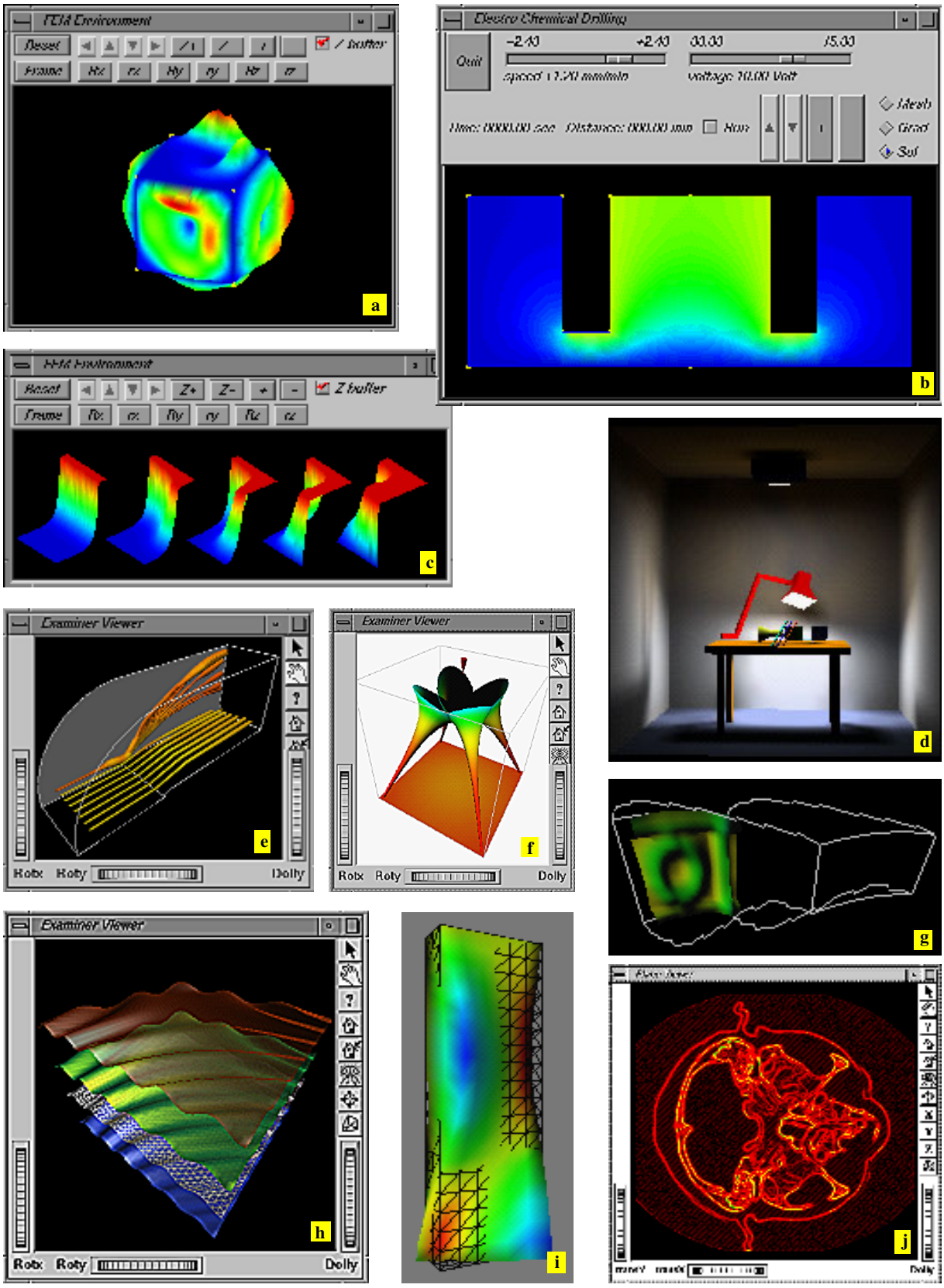

Figure 7: Visualizations and simulations performed in the VISSION environment