Oorange: A Virtual Laboratory for Experimental Mathematics

C. Gunn, A. Ortmann, U. Pinkall, K. Polthier, U. Schwarz

May 15, 1997

Abstract

Oorange is a virtual laboratory for experimental mathematics. It consists of a set of infrastructure services supporting the creation, execution, and dissemination of mathematical experiments. For each component of a traditional physical experiment, there is a corresponding *Oorange* infrastructure feature:

* **Object of study:** High level software classes
* **Laboratory equipment:** Foundation software classes and function libraries
* **Configuration of specific experiment:** Computational network composed of objects
* **Monitor and control:** Object inspection; 2D and 3D viewers
* **Running the experiment:** Animation objects
* **Recording the experiment:** Archiving and scripting
* **Disseminating result:** Documentation

A hybrid language scheme underlies the design: interpreted scripts in *Tcl* manage tasks requiring high flexibility, while a compiled object library in *Objective C* supports the underlying mathematical algorithms. The resulting system is intended to be accessible to wide range of expertise levels. *Oorange* is free software distributed according to a GNU-like license agreement.

1 Introduction

Experimentation has a distinguished tradition in mathematical research. Experiments often play an important role in formulating new conjectures and discovering paths toward the proof of such conjectures and of other results. As computational resources have become more powerful in recent years, this domain of activity is growing into a mathematical field in its own right, called *experimental* mathematics. There is a perceived need for a standardization so that experiments made by one researcher can be repeated or modified by others with minimal effort.

The *Oorange* project is a response to this challenge. It is an initiative of the Sonderforschungsbereich 288 of the Technical University Berlin, "Differential Geometry and Quantum Physik". The project development commenced in September 1994, the first beta version was release Christmas 1995, and Version 1.0 is slated for release in October 1996. We postpone an account of previous work until Section 5 in order first to introduce the terminology in which comparisons and historical influences can be expressed.

The key impulse behind the project is to create a framework for experimental mathematics that is appropriate to the contents of this science. Since the field of experimental mathematics is modeled on the traditional models of physical sciences, we found it useful to maintain the correspondences to that realm. With this in mind, the following seven-fold subdivision of a mathematical experiment is helpful:

* **Object of study:** High level software classes
* **Laboratory equipment:** Foundation software classes and function libraries
* **Configuration of specific experiment:** Computational network composed of objects
* **Monitor and control:** Object inspection; 2D and 3D viewers
* **Running the experiment:** Animation objects
* **Recording the experiment:** Archiving and scripting
* **Disseminating result:** Documentation

The correspondence to traditional experimental practice is only approximate; significant differences arise from the non-physical nature of software. For example, there is a further level of design necessitated by the fact that there is no physical laboratory to hold all these components. *Oorange* is accordingly organized as a virtual laboratory, which presents a unified user interface integrating all the above components.

In order to motivate the detailed discussion of the components we first introduce the virtual laboratory.

2 Overview of Oorange

This description will proceed without technical details to give an overview of how *Oorange* is used. Terms which are used without definition will be printed in *italic* font. They will be explained later on in the more detailed description of each component.

Refer to the screen snapshot of an *Oorange* session You should see four windows labeled: Network Manager, Inspector Manager, Scene Viewer, Script Editor.

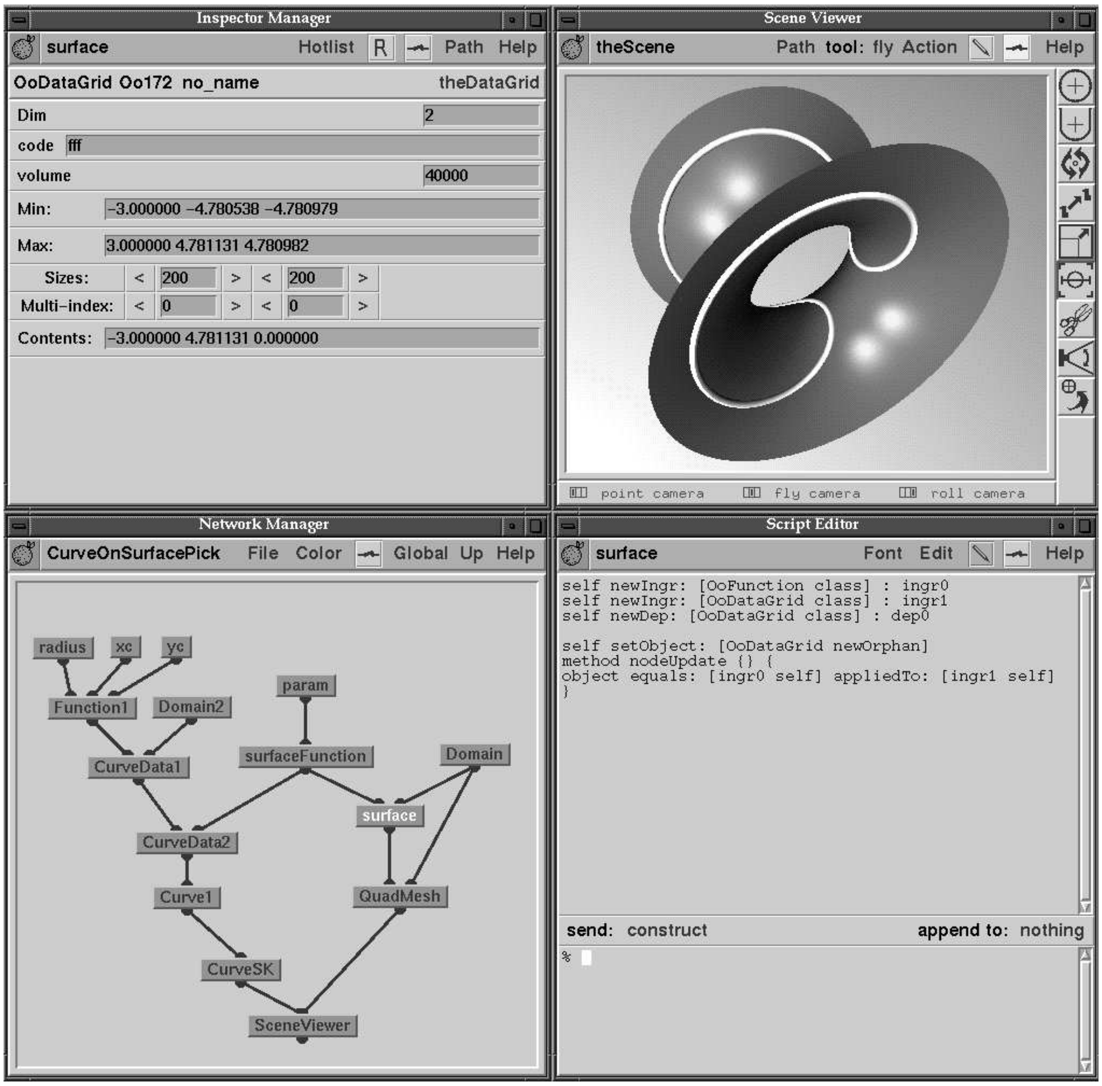

Figure 1: A screen snapshot showing the major components of the *Oorange* frontend. *(See also Color Plate on page 19.)*

This image represents the view of *Oorange* upon loading the *node file* `Curve-OnSurfacePick.nod`. The upper right window displays a 3D scene featuring a parametric surface on which a curve has been overlaid.

Begin by observing the *Network Manager* (see Section 3.2 and [12]) in lower right window of Figure 1. You see the graphical representation of the computational network which has been loaded. When *Oorange* is started, this window is empty. Users can assemble networks by hand, node by node, using the tools in the network manager; or they can load predefined nodes, as in this case.

Each icon in the window represents a *node*; a node is a unit of computation. Each node contains an *object* or *sub-network*. An edge joining the bottom on one icon to the top of another represents a *dependency* of the second on the first. Whenever any node lying above is changed, then all nodes which are below that object are updated to reflect the changed status. This *update* process is one of the fundamental infrastructure services of *Oorange*.

The Network Manager contains only the graphical representation of the network; using other components the actual contents of nodes can be examined and adjusted.

At any time, the Network Manager has a *current selection*. In this case, it is the node labeled `surface`; its label is white. The *script* of the current selection is always displayed in the *Script Editor* (see Section 3.2.1 and [12]) to the right of the Network Manager. This script contains a complete description of the node and the state of its objects. These parts are generated automatically as the network is assembled. This script is also the most common way to customize the behavior of the node. For example, the `nodeUpdate` method defines the commands which are invoked whenever the node is updated. Since the language of the script is interpreted, the user can edit any part of the script interactively and see the results immediately.

The current selection also appears in the *Inspector Manager* (see Section 3.3) in the window above the Network Manager. In our example, the object within the `surface` node is being inspected. By examining the Inspector Manager window, you can see that it's an object of class *OoDataGrid*, and also can read off more information about its current state. Using this inspection panel you can edit the contents of the datagrid; the picture automatically changes to display the updated data. That is, the various panels of the virtual laboratory are synchronized so that changes made in one component are propagated to others.

Finally, return to the *Scene Viewer* (see Section 3.3.2) in the upper right window. It's really also part of the inspection process, but it's such a large and self-sufficient part that it has been split off and given its own top-level window. The viewer is an interactive 3D viewer [12]; on the right side of the window are icons for choosing among the available interactive tools.

With this quick overview of how a running *Oorange* session appears, we proceed to more detailed description of the underlying components.

3 Components of the Virtual Laboratory

3.1 Software Ingredients

We begin with a sketch of the programming languages used in *Oorange* and how they are related to one another.

3.1.1 Objective C

The primary language to represent mathematical objects (as *experimental* object of study) is *Objective C* [1, 2]. The choice of an object-oriented language can be justified by the fidelity with which these languages mirror the hierarchical nature of mathematical structure. However the choice of *the* object-oriented language is a subject of debate.

In contrast to its better known brother *C++*, *Objective C* offers a genuine *run-time system* that allows new classes or new methods to be defined during a running session [1]. This feature will be invoked often in the ensuing discussion so that the wisdom of our decision will be hopefully established.

Oorange comes with over 200 *Objective C* classes. The majority of them at this time are related to infrastructure functions. Many of these classes will be introduced in the discussions of the specific components below. An increasing number of more mathematically oriented classes are also in the process of construction. See section 4.1 for a discussion of two of the most useful foundation classes.

Objective C offers an alternative to subclassing via the use of *protocols*. A *protocol* is a simply a set of method declarations; any class can choose to implement any protocol. For example, *Oorange* uses protocols to define animation, archiving, inspection, and 3D geometry behavior. Participation in the corresponding infrastructure service is then independent of class hierarchy.

There are also provided with *Oorange* a wide set of C libraries of functions supporting mathematical experiments. These are provided as functions rather than objects in the interests of performance; some of these have object analogs when ease of programming is more importance than raw performance.

3.1.2 Tcl

The other main language used in *Oorange* is *Tcl* [7, 14]. Roughly speaking, a *Tcl* layer handles all user interaction. As an interpreted language, *Tcl* is ideally suited to rapid prototyping and flexible configuration. The coordination of *Tcl* and *Objective C* is, as described below, one of the strong points of the design.

Oorange incorporates several *Tcl* packages or extensions to handle different needs. *Tk* is used to create and update all GUI widgets. Other packages are used to establish connections to the *Objective C* layer. *iTcl*, in conjunction with *Tk*, is used to provide a class hierarchy of inspector panels that mirrors the *Objective C*

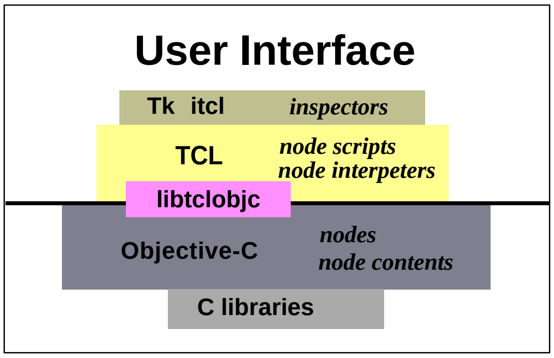

Figure 2: Schematic representation of relation of *Tcl* and *Objective C* in *Oorange*

class hierarchy. *libtclobjc* [4] is a C library that allows *Tcl* interpreters to interact transparently with *Objective C* objects and vice-versa [12]. Each *Objective C* class and instance appears as a command to the *Tcl* interpreter. Users can then issue class or instance method calls to the *Tcl* interpreter, which parses them and dispatches them directly through to the *Objective C* runtime system. There is no need to register classes or methods with interpreters; this information is obtained "for free" by querying the *Objective C* runtime system. *Oorange* extends this facility so that references to classes which are not yet loaded will trigger dynamic loading of the required library. This interplay between *Objective C* and *Tcl* (Figure 2) is one of the most powerful features of *Oorange* and runs as a thread throughout the following discussion.

Users are not expected to be proficient in *Tcl*; it is an *advanced* skill.

3.2 Computational Network Manager

Computation networks [6] are a popular programming paradigm for experimental mathematics. The nodes of these networks each represent some computational activity. For simplicity we call this the node's *action*. Each node may have a set of *inputs* and *outputs*; the inputs may be thought of as arguments to the computation and the outputs as the results. Whenever the inputs to a node change, then the node's action is invoked to bring the outputs up to date.

The nodes are assembled into a directed graph by adding edges, each of which connects an output of some node to an input. This establishes a *dependence* of the second node on the first. Whenever the outputs of the first node change, the second node's action must be invoked. Implementations typically provide an automatic algorithm which propagates such changes along the edges of the graph and *updates* the nodes.

Cycles in the graph must be handled carefully by the update mechanism, since they can easily lead to infinite loops. Most implementations have a strategy to allow cycles; the computational action of some node is expected to apply a conditional that terminates the loop in the update process. We return to this below when discussing how *Oorange* handles this problem.

An important feature of any implementation of such computational network is what exactly is passed from node to node. In *data flow* networks the system manages the movement of data during the update process, typically by copying the data along the edges from output to input. This model has proved to be very successful when applied to a wide range of natural scientific domains. The commercial packages AVS [13] and Explorer [3] are both based on this model.

However the situation with respect to mathematical research is not satisfactory. The disadvantages of existing data flow implementations for mathematical experiments are essentially twofold. First, there is a restricted set of data types which can be moved along an edge. Most actual mathematical structures do not fit into these simple types. Second the action of a given node is typically difficult to modify. Update actions are typically written in a compiled language such as C or Fortran; when it possible at all to modify them, such modifications require advanced programming skills, compilers, and source code. In practice this restricts many researchers to using actions which are not exactly what they want, or starting again from scratch and creating a new one.

Oorange has maintained the essential update mechanism from the data flow model but has replaced the fixed data types and fixed actions with more flexible tissue. To begin with, *Oorange* concentrates on *objects*, rather than *data*. The contents of the simplest *Oorange* nodes are *Objective C* objects. Object pointers are passed from one node to the next along the edges of the graph, freeing the user to pass arbitrary types of data between nodes.

Secondly, rather than having a compiled, relatively fixed update action , an *Oorange* node is provided with an editable *Tcl* script, the *node script*. This typically consists of a sequence of *Objective C* statements (as explained above in 3.1.2) directed at the contents or ingredients of the node. This script can be modified, or new *Objective C* commands may be selectively executed before adding them to the script. Node scripts combined with network nodes provide an elegant solution to simulating computational loops, such as iterated convergence algorithms.

The *Oorange* network model, then, provides the flexibility and power that the experimental mathematician needs to carry out his experiments.

How are *Oorange* networks constructed? They can be interactively assembled in the *Oorange* Network Manager. This provides a visual "assembler" for creating new nodes, loading existing subnetwork nodes, adding ports and links, cutting and pasting, and a variety of other useful editing tasks.

3.2.1 The script editor

The original impulse to use *Tcl* in *Oorange* networks was to provide editable update actions. This proved so effective that a much wider application of *Tcl* within the network manager eventually took form. In fact, *Tcl* scripts became the archiving medium for *Oorange* networks. As users construct or edit networks in the network manager, a *Tcl* script is generated which describes the state of the network as a sequence of *Objective C* commands (using *libtclobjc* – see Section 3.1.2). Because these scripts play such an important role in the experimental process, they have been separated out from the standard inspection process and given their own top-level window, the Script Editor.

In the script editor the user can see a complete description of the currently selected node and the state of its objects. For example, in the figure above, the currently selected node is *Surface*; its script appears in the script editor to the right of the network manager.

Among the statements generated automatically in the network editor are those which describe the dependents and ingredients, the links, and the contents of the node. Others are provided by the node developer, in the form of *Tcl* procedures which provide *customized* node behavior. The `nodeUpdate` method is probably the most important of the latter, but there are others which are invoked, for example, in connection with animation, 3D picking, and node documentation. These scripts provide a powerful prototyping facility where ideas and algorithms can be developed and tested.

At the bottom of the script editor window is a shell where the user can type commands directly to the *Tcl* interpreter associated to the node. These commands are immediately executed.

It is important to note that it is possible to be a productive user of *Oorange* without learning *Tcl* or using the script editor. There are many predefined nodes and networks which can be hooked together without having to edit the attached scripts at all. And the subset of *Tcl* which is used in the scripts is almost identical to *Objective C*.

3.3 Object Inspection

All objects in an *Oorange* network are subject to inspection [12]. The network editor maintains a current selection, and this node is always inspected in the *Oorange* inspection manager. The inspected object is not always a node, as the following discussion will make clear.

The inspection manager queries the inspected object for its inspection command. This is expected to be a *Tcl* command which when executed will yield a set of panels of *Tk* widgets that represent the state of the object. The inspection manager then controls the display of these panels. In order to take advantage of the object hierarchy, *Oorange* implements its object inspectors using *iTcl* package [5], an object-oriented version of *Tcl*. In this way, a class hierarchy of inspectors is built which mirrors the class hierarchy of *Objective C*.

Normally, exactly one panel is displayed at one time in the inspector manager. If the user desires to have a particular panel persist, he can add it to a "hot-list" which contains a list of commonly inspected panels. This avoids the proliferation of panels that, like a littered laboratory bench, can sabotage the experimental task.

The inspector for a class allows the user to inspect the state of the instance variables for a given instance of the class, and when appropriate, to edit that state. The inspection process depends on whether the instance variable is itself an instance of an *Objective C* class (see Section 3.3.1).

Any editing performed on an object within the inspector manager will trigger the network update mechanism. This feature can be temporarily disabled if several edits are desired before update occurs. Editing commands can be automatically appended to the script of the node if a record of the edit is desired.

There is an *Oorange* protocol *OoClassFields* which frees the programmer from writing his own inspectors. Classes which conform to the protocol will receive a default inspector which they can customize as required.

Another feature of the *Oorange* inspection process allows developers of complicated nodes to make the node appear like a simple node to the inspection process. This is achieved by so-called *file cabinets* which encapsulate in a single flat list, all the interesting objects contained anywhere within the sub-network lying inside the node. For example, Figure 3 shows the file cabinet interface to the standard SceneViewer node. Developers who wish to add this feature to a node are only required to define a `fileCabinet` method in the node script.

Figure 3: File Cabinet of the standard SceneViewer node

It is also possible to create inspection panels directly in the node script without basing them on a *Objective C* object. Such panels are called *private* panels. For example, a *Tcl* variable rather than an *Objective C* object may be used to control the behavior of a node. It is possible then to create an inspector panel which allows editing of the value of this variable by implementing an `inspect` method for the node script.

3.3.1 Navigation in the inspection manager

Once the current node is inspected, it is possible to navigate within the inspector manager in three main ways:

* **down:** Inspect a sub-object
* **lateral:** Go to a different panel of the same object
* **up:** Return from inspection of a sub-object

The inspector manager maintains two menus to perform this navigation. The first presents a list of *down* and *lateral* options available from the current object. The second keeps a record of the sequence of sub-objects chosen with the *down* option, so that the user can pop back up to any desired level at any time.

3.3.2 Viewers

One further form of inspection is provided by *Oorange* which deserves mention. Objects which consist of 2-D (3-D) data can be inspected with a 2-D (3-D) viewer provided with *Oorange*. These viewers share a common protocol, *OoToolProtocol*, which describes an tool interface for handling mouse down/drag/up events.

The 2-D viewer [12] supports a wide variety of operations on 2-D images, including arbitrary resizing and translation, and reading and writing a wide variety of file formats. The underlying object is the general *OoDataGrid* class (see Section 4.1).

The 3-D viewer [12] is supported by a large class library in major features similar on the Inventor class library originally from SGI. There is a device-dependent core which currently supports OpenGL (immediate and offscreen modes) and *mentalray* (a commercial ray tracer). Point sets can be either 3 or 4 dimensional, of float or double type. There are classes for cameras, lights, appearances, materials, fog, textures, drawables, bounds, pick actions, and transforms. Every viewer must contain a scene, which is the root of a scene graph describing the scene. Scenes contain a drawable, a camera, a device, and several stacks. There are a variety of shape related classes. The base class is shape group, which has a list of children. Shape kits have additionally a transform, an appearance, a material, a texture, and a texture transform. Shape instances have a list of transforms. Finally, there are geometric types including indexed face sets, quadrilateral meshes, indexed line sets, triangulations, and cube/cone/sphere/tori classes.

Interaction in the 3-D viewer is provided through a variety of standard tools, which can be used to select a particular shape and rotate/translate/scale it; or to move the camera. There is a *wild card* tool which distributes the current pick information to the scene graph. A user can create a customized tool at a node simply by adding `mouseDown`, `mouseDrag`, and/or `mouseUp` procedures to the script (analogous to the nodeUpdate method). For example, in the node shown in Figure 1, a `mouseDrag` procedure allows the user to drag the curve *by its center* around on the surface. More sophisticated uses are easy to imagine and implement.

3.4 Time

One of the essential elements present in the physical laboratory which must be recreated for the virtual laboratory under discussion here, is *time*. Its importance in the overall design can hardly be overstated. Every experiment runs its course in the river of time. Experiments are described, controlled, and recorded in terms of the passage of time. In order to provide a convincing simulacrum of the real thing, time has a special status within the *Oorange* design [12].

Time has its own distribution system based on a tree of time managers. Each time manager is attached to a specific node and is responsible for all animated objects in the node or its children. Compare [9] for a similar time concept.

Objects can participate in the flow of time by conforming to one of several animation protocols, (each of which is a prerequisite of the next):

* **OoAnimated** defines methods that apply to any object that is interested in time.
* **OoValueAnimated** extends the **OoAnimated** protocol for objects that have values that depend on time.
* **OoKeyAnimated** extends the **OoAnimated** protocol by allowing manipulation of key frames.

When a new instance of an animated object is created, it has the responsibility to register itself with the *time manager* of its containing node. Time managers relay *animated* methods to the *animated* objects registered with them.

The current *Oorange* class library includes animated classes for numbers (floats and doubles), vectors, colors, and linear transformations. The vectors can be elements of the classical spaces E^3, H^3, S^3 or P^3 (euclidean, hyperbolic, spherical, or projective 3-space). Euclidean similarities are decomposed into appropriate factors which are then interpolated. All interpolation currently is linear.

The design philosophy of *Oorange* is to avoid using animated variables as instance variables of objects; instead, use ordinary variables (floats, vectors, etc) as instance variables and use networks to connect animated variables to these variables. This simplifies the code of objects and lets the network update mechanism take care of keeping all variables synchronized to the current time.

3.4.1 The hierarchy of time managers

Time managers themselves are animated objects and register with the time manager of their "father" node. This results in the creation of a *shadow* hierarchy of time managers. There are no lateral connections between time managers. On first sight, this profusion of time managers might seem to be an extravagance of questionable value. However, the advantages of the hierarchy have become clear as *Oorange* has matured:

local time Time can be dilated and translated for a given sub-tree of the hierarchy. Or, time related actions can be restricted to a sub-tree by concentrating attention on the time manager based at that sub-tree. Key frames can be restricted to a sub-tree, or animations can be played back only for it.

nested time This is analogous to the minute and second hands of a watch. In many computational processes there are actually such nested forms of time. For example, a surface may be built of a curve swept across space. The second hand, sitting in a time manager within a sub-network, would control the generation of one curve, while the minute hand, sitting in the time manager of the "father" node, would control the accumulation of successive curves into one surface.

3.4.2 *Tcl* Scripts and time

Nodes can participate in the time flow even if there are no *Objective C* animated objects within the node. It is possible to force the time manager to be created and activated. Then, the update script of the node can query the time manager as to the state of the animation and take appropriate action. For example, this is how movies are recorded from the output of the viewers. If there is special actions to be taken at the beginning or end of an animation, then the node should provide procedures named *beginAnimation* and *endAnimation* to perform these actions.

3.5 Archiving

One of the goals of *Oorange* development was to avoid creating another file format if possible. This goal has been partially reached, in the following sense: Instances of *Oorange* objects archive themselves as *Tcl* scripts containing *Objective C* method calls (see Section 3.1.2). That is, the format of the archive is *implicit* in the definition of the object rather than being imposed from outside.

To be exact, the sequence of the archiving method calls is governed by the *OoClassFields* protocol. This protocol was mentioned in 3.3 above with respect to inspection. As there, the situation here requires that a class provide a description of its instance variables. In this case, the protocol provides information from which the set/get methods for all public, "archivable" instance variables, can be generated. Then, for each such pair, the get method is invoked by the *Tcl* interpreter and yields a string which is then appended to the set method name to yield an *Objective C* statement which is appended to the archive file. Here we skip over various subtleties described in more detail in the protocol documentation.

The resulting archive is in ASCII form, so that experiments can be edited and exchanged in a human readable form.

The possibility of including predefined network nodes within a larger network presents a serious challenge for the archiving process, since users need to be able to save changes to an included node without losing the reference to that node. *Oorange* provides a solution to this problem by recording a set of *change commands* along with the reference to the included node.

3.6 Documentation

Clear and ubiquitous documentation is central to the *Oorange* philosophy. To begin with, all *Oorange* documentation exists in HTML format, and is directly accessible from within *Oorange*. There are several levels of documentation available:

∗ **Tutorials:** There are currently nine tutorials describing various aspects of the *Oorange* system. These are intended for new users and cover the topics of overview, network editor, the *Tcl/Objective C* connection, 3D viewing, 2D viewing, animation, inspection, datagrids and functions, and adding new classes to *Oorange*. These can be loaded from *Oorange* into an HTML browser.

∗ **Class documentation:** *Oorange* provides a documentation extraction system "Objective-Doc" [12], which acts on class interface files and source files to generate LaTeX or HTML files containing documentation including the following features:

  – automatic generation of class hierarchy above this class with links
  – class description
  – class and instance method names
  – instance variables
  – insertion of links to other classes or methods

  Protocols and categories are handled similarly. The resulting documents are available through a Web-based class browser (Figure 4).

∗ **Node documentation: annotate** methods (analogous to the nodeUpdate method) can be attached to any node script to provide node customized documentation for the node. This can be as simple as a character string to be displayed above the node icon, or an arbitrary HTML document can be loaded into the browser of your choice.

∗ **Search capability:** The Web-based documentation is equipped with a general search capability.

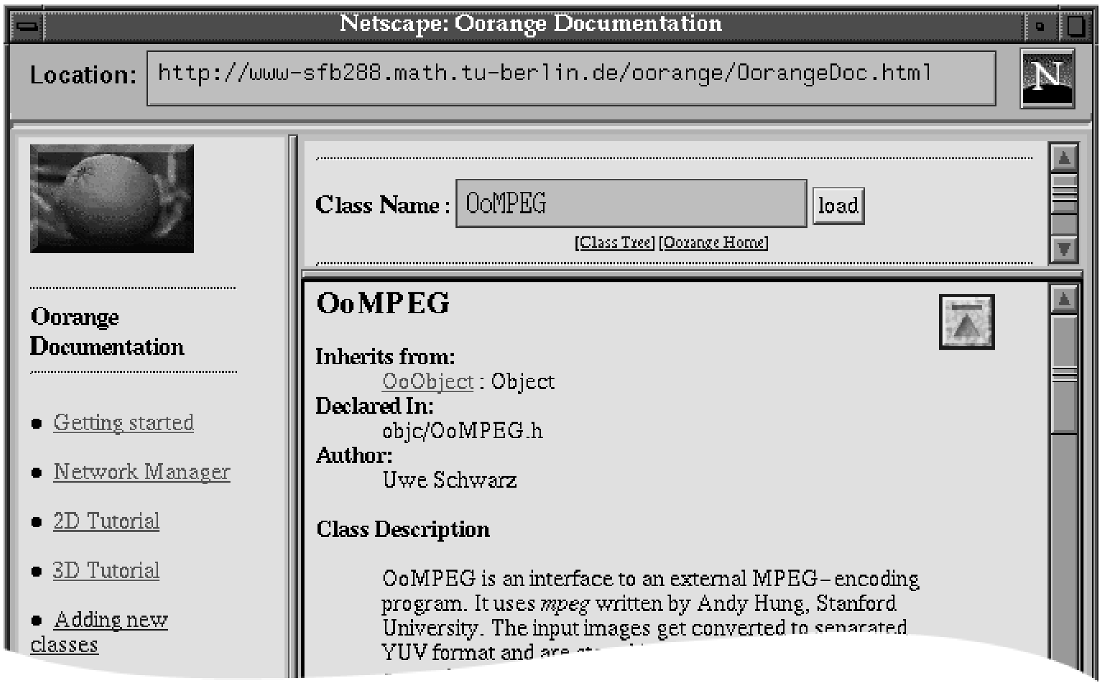

Figure 4: The class browser of the *Oorange* Online documentation

4 Special Tools

4.1 Workhorse Classes: **OoDataGrid** and **OoFunction**

Rather than try to describe the class hierarchy provided by *Oorange*, we limit our discussion to two foundation classes which are part of many *Oorange* networks [12]. The first, *OoDataGrid*, represents arbitrary multiple dimensional arrays with an arbitrary "fiber" of data at each entry. This fiber is represented as a character string (adopted from *Objective C*) with one character for each entry. For example, "ddd" represents a chunk of data consisting of three double precision numbers, while "difz" represents a sequence of a double, integer, float, and complex packed together.

The datagrid class is the class underlying images and shapes. The currently selected node in the *Oorange* session image contains an instance of this class; the inspector for the instance is highlighted in the inspector manager above the network editor. There are a wide range of operations defined on OoDataGrid, such as resize, convolution, fiber re-mapping, fiber conversion, grid reformatting, and contraction. Datagrids can be combined by binary arithmetic operations, tensor product, or appending; or extracted by slicing.

Closely coupled with datagrids is *OoFunction*. This is an object wrapper for a C-function with arbitrary input and output, which can be edited, compiled and dynamically linked into a running *Oorange* session. In this way users can usually adapt existing C code to run in *Oorange* without having to learn or create any *Objective C* code. In particular, the full power of standard C libraries can be harnessed within *Oorange*. Functions can be applied to datagrids to yield an image datagrid.

4.2 Connectivity to Other Programs

Using the *Tcl expect* package it is possible to establish dialogs within *Oorange*, with other running programs. This package facilitates sending commands to remote programs and returning results. The program must accept some sort of interactive command stream. This has been succesfully done with *Mathematica* [16], and refined to the point that arrays generated in Mathematica can be converted to *OoDataGrid*'s, and vice-versa, surfaces generated in *Oorange* can be shipped to Mathematica to generate Postscript output. This is a very good example, since the strengths and weaknesses of the two products are well-matched: *Oorange* provides an articulated object-oriented structure that Mathematica lacks, while Mathematica can be invoked for symbolic computation. A fitting image of the relationship is that *Oorange* is a sturdy tree and Mathematica a fruitful vine climbing upon it.

5 Previous Work

Oorange has borrowed and/or inherited features from a variety of software products, each of which addressed in some way the challenge of the virtual laboratory. One of these influences is GRAPE [11], a mathematical programming environment of the SFB 256 at the University of Bonn. GRAPE pioneered an object oriented approach similar to *Objective C* in connection with mathematical visualization. The focus on data objects was extended in *Oorange* to the concept of a dependency graph of programmable objects.

Geomview [8], a 3D visualization tool from the Geometry Center, features the concept of an *external module*, an independent computational unit which feeds geometric information to the viewer and vice-verse. This concept is similar to the *node* in *Oorange*. *geomview* lacks however the ability to create network graphs from nodes; the default graph is a star with the viewer at the center and external modules on the periphery. AVS, as described above (Section 3.2), is a ancestor of *Oorange* and much of the network design was a response to its perceived shortcomings.

Open Inventor [15] had a strong influence on the design of the 3D classes. However, *Oorange* chose a more flexible node model than Open Inventor's, in which the node and its contents are more separated. New node contents in Inventor is achieved typically by subclassing the node class. *VTK* [10] deserves mention here even though it has no influence on *Oorange*, since it was developed simultaneously and independently. It offers a *Tcl* interface to a large *C++* class library focused on 3D visualization tasks. There are AVS-like data flow classes.

The inspection component of *Oorange* is, to our knowledge, more sophisticated than the analogous services offered in other existing systems. The integration of the animation component with the network update mechanism appears to be an original contribution.

6 Future Directions

There remain some unresolved design issues. The connection of *Tcl* and *Objective C* layers has been a theme throughout the development process. The primary challenge is knowing exactly how to subdivide the tasks between the two languages. The dividing line cannot be hard and fast; some tasks prototyped in *Tcl* will naturally migrate to *Objective C* after they have been proven. But even after this has been factored out it is not always clear where to make the division. threads, garbage collection, error handling

Some unsolved problems arise from the extremely loose coupling between *Oorange* nodes and their contents. This is a source of strength, but also makes it hard sometimes to be efficient. In Open Inventor for example dependencies between objects are registered and so dependent objects can be easily identified; while in *Oorange* dependencies may be defined in node scripts and may influence only some of the instance variables of the dependent object, making this identification difficult if not impossible. As a result, dependencies in the network graph are not consistently enforced in the inspection manager. That is, users are allowed to edit objects which in reality are controlled by upstream objects. The result is that node update will overwrite the user's editing.

Finally, one of the original but not-yet-implemented goals of *Oorange* is to use the "distributed objects" feature of *Objective C* to instantiate nodes on a network. The node design of *Oorange* should make this easy to do, as long as developers observe the convention that a node represents an independent unit of computation.

7 Conclusion

With *Oorange* we have endeavored to create a virtual laboratory for conducting mathematical experiments. Our strategy was to design and implement infrastructure services that correspond to the component elements of the experimental process, and then to assemble these components into a seamless whole. We have confirmed the essential features of the design. We consider the following features to be particularly important advances on the existing solutions:

* Synergy of *Tcl* and *Objective C* layers allowing easy prototyping and gradual learning curve without sacrificing high performance.

* Object– rather than action–based network model.

* Ubiquitous presence of time in the model.

* Easy integration of existing resources (either as C–code or as co–executable).

With the distribution of Version 1.0 to the general public we hope that a wider user community will find its way to join the development outlined above.

Acknowledgements

Other members of the *Oorange* development team, both current and past, include David Oliver, Axel Friedrich, Pat McDonough, and Markus Schmies.

Appendix: Availability

On the level of hardware, *Oorange* currently runs on SGI machines and Linux-based PC's. In the near future we plan to port to Solaris platforms also. *Oorange* is free software distributed according to a GNU-like license agreement. A binary version may be obtained from one of several *anonymous ftp* sites.

Up-to-date information about *Oorange* and its distribution is available at the World Wide Web server of the Sonderforschungsbereich Differential Geometry and Quantum Physics: *http://www-sfb288.math.tu-berlin.de*.

References

[1] Addison-Wesley Publishing Co., Reading, Massachusetts, *NeXTSTEP Object Oriented Programming and the Objective C Language*, 1993, ISBN 0-201-63251-9.

[2] B. J. Cox and A. J. Novobilski, *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991. ISBN 0-201-54834-8.

[3] M.-A. Halse, *IRIS Explorer User's Guide*, Silicon Graphics Inc., Mountain View, California, 1993.

[4] R. A. McCallum, *Libtclobjc*, 1994.

[5] M. J. McLennan, *[incr Tcl]: Object-oriented programming in tcl*, Technical report, University of California, Berkeley, 1993.

[6] A. Ortmann, *Modellierung von Abhängigkeitsgraphen*, Studienarbeit, Technical University Berlin, Feb 1996.

[7] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Professional Computing Series. Addison Wesley, 1994.

[8] M. Phillips, *The Geomview User's Manual*, The Geometry Center, 1993.

[9] K. Polthier and M. Rumpf, *A Concept for Time-Dependent Processes*, in: M. Göbel, H. Müller, and B. Urban (eds.), Visualization in Scientific Computing, Springer Verlag (1995), 137–153.

[10] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics*, Prentice Hall, 1996.

[11] Sonderforschungsbereich 256, Grape *Manual*, University of Bonn, Sept. 1995, http://www-sfb256.iam.uni-bonn.de/grape/main.html.

[12] Sonderforschungsbereich 288, Oorange *Online Manual*, TU Berlin, http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html.

[13] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, The application visualisation system: A computational environment for scientific visualisation. *IEEE Computer Graphics and Applications*, 9 (1989), 30–42.

[14] B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, 1995.

[15] J. Wernecke, *The Inventor Mentor*, Addison Wesley, 1994.

[16] S. Wolfram, *Mathematica*, Addison Wesley, 1991.

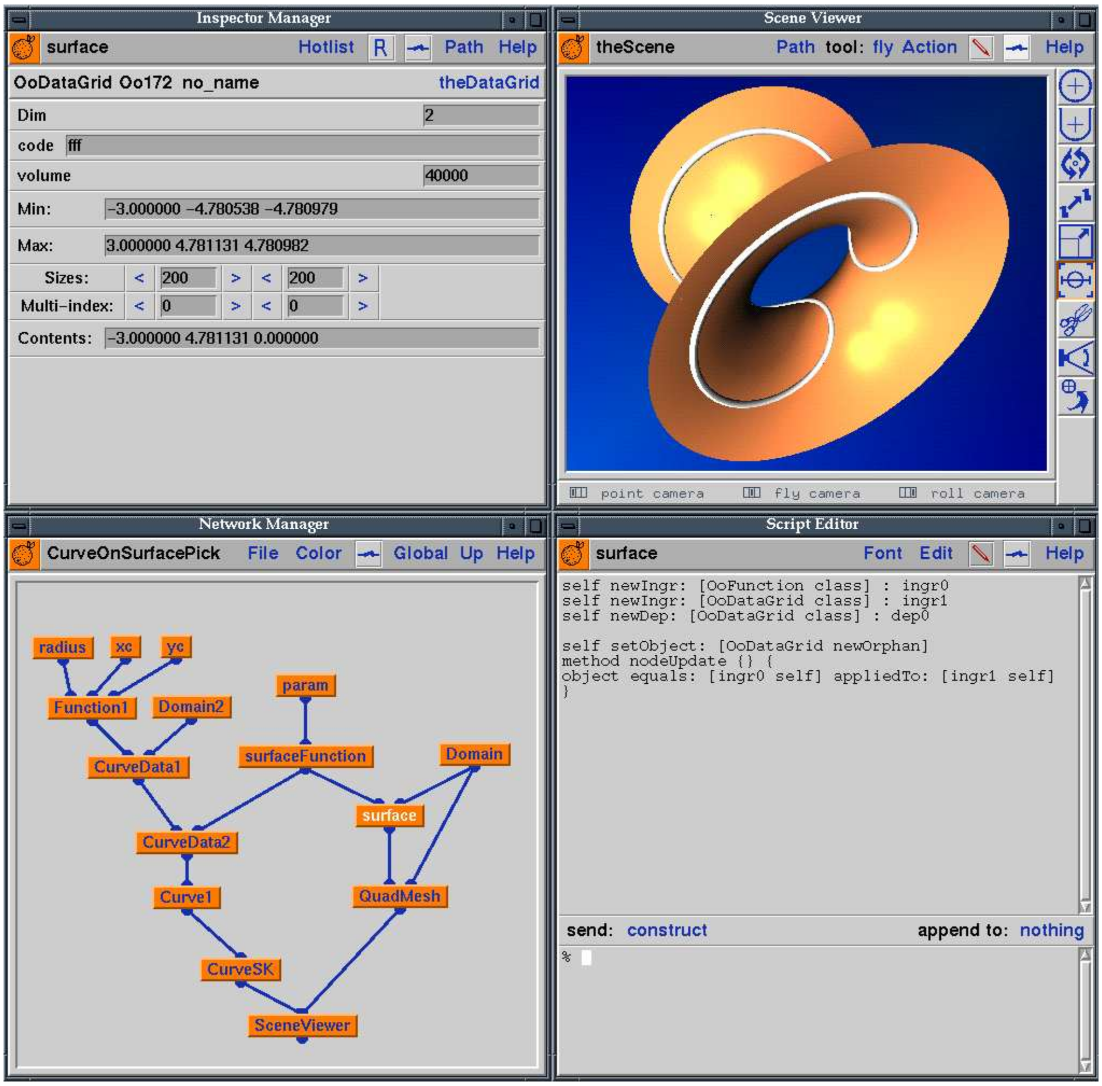

Figure 5: Oorange is an extensible object-oriented visualization environment for mathematical experimentes. A design goal was to allow easy programming of the system with different levels of expertise ranging smoothly from novice to experts. The components of the Oorange frontend are the Network Manager, Script Editor, Scene Viewer and Inspector Manager.