Strategies for Scientific Prototyping in Smalltalk

Sandra S. Walther, Center for Computer Aids For Industrial Productivity,
Richard L. Peskin, Dept. of Mechanical and Aerospace Engineering,
Rutgers University, Piscataway, NJ 08855 - 1390 *

Abstract
This paper describes the design of a scientific prototyping environment in Smalltalk and discusses implementation strategies to achieve high performance interactive modeling of computationally intensive physical problems. Classes for scientific visualization, including contour plotting and 3D surface representations which incorporate the model-view-controller paradigm are presented. Techniques for inclusion of user primitives written in C to support computationally intense methods are described in detail in their current implementation on advanced Smalltalk workstations (SUN4, Ardent Titan, Tektronix 4317).

1 Introduction

Scientific computing, notably numerical modeling and simulation, is anchored in a thirty year old software tradition. It is now documented that this computing methodology is hindering computation as a tool for experimentation and discovery [1]. While parallel computers and supercomputers are advancing the potential for faster scientific computation, the cumbersome software environments provided for these facilities tend to be a limiting factor to effective use. For the past three years we have been developing an alternative to the traditional environment; for an overall review of the program see [2].

Specifically, we are designing an interactive system that will promote rapid prototyping of new numerical models and the analysis of these models. Since the objective of the scientific simulation process is to model the behavior of the physical system, it is natural to select an overall software environment which is intrinsically behavioral, i.e. an object-oriented environment. Smalltalk-80 provides a mature object-oriented environment, is fairly complete with tools for scientific computing (numerical classes, graphics, etc.), and offers a potential alternative to traditional scientific computing software. As important, Smalltalk greatly accelerates our own development process.

Thus, we are constructing an interactive environment for scientific computing based on Smalltalk-80†. There are many functions such a system must perform, including automatic translation of user based problem specification (e.g. partial differential equations) to run-time code, interactive graphical input and output, interactive model modification, etc. This paper will concentrate on Smalltalk implementation strategies for two of the important functional areas, graphics and model prototyping. Areas such as the knowledge-based automatic programming sub-system (actually written in Prolog) are described elsewhere in the literature [3]. Of primary concern are the strategies employed to effect a balance between the need to maintain a high level of user accessibility to his scientific model classes and the need to maintain an acceptable interactive performance level. This latter requirement often demands integrating Smalltalk with higher performance environments through addition of user primitives, interfaces to "back-end" high performance (e.g.parallel) computers, or a combination thereof.

2 Interface for Scientific Modeling

2.1 Scientific Visualization Classes

The first prerequisite of a scientific interface is the capacity to represent numerical information in traditional graphic formats, namely, two dimensional xy plots, contour plots, and three dimensional surface plots. The numerical information is computed in floating point (double or single precision) and may be conveyed to a graphing utility directly as it is computed or it may be read in as a file after being downloaded from a separate computing facility. Ideally, the user should be able to manipulate the visualization interactively. Prior to adopting Smalltalk as its visualization environment, the graphics interface group had been developing libraries of device-independent 2D and 3D

```
VisualizationObjects        DisplayBehaviors          Controllers
   Models)                      (Views)

Object                      View                      MouseMenuController
   DataModel                  DataView                    DataViewController
     DataSurface                 ContourView                 ContourViewController
          DataContour               LogisticEquationView
          PrimitiveSurface       DataSurfaceView             DataSurfaceViewController
              Flag                  FlagView                    FlagViewController
   LogisticEquation
   Oscillator
```

Figure 1: **Graphical Interface Classes for Scientific Modeling**

graphics functions, written in C, which formed the basis of a set of interactive graphing tools for scientific and engineering data analysis (xytool, streamtool, contourtool, surfacetool, modeltool) that were implemented under SunViews and Xwindows on a variety of workstations and as a "front end" to several parallel computing facilities. [5] While not yet object oriented in implementation, each tool was organized logically around a set of *visualization behaviors*; the 2d graphing methods knew how to present xy data according to the Graphics Kernel Standard; the 3d graphing methods knew how to present xyz data, using standard transformation and perspective techniques [6],as three dimensional shapes. These behaviors were device independent in the sense that they were entirely encapsulated from the drawing and display functions through which the consequences of their operations were made visible. They were decoupled from the event management facilities of the interfacing environment (SunViews or Xwindows) as well. This functional distinction between the graphic object (the data), its manifestation as a drawable, displayable entity in a particular device environment, and the user control mechanisms (mouse and/or keyboard interplay with the ongoing representation) mapped easily to the model-view-controller paradigm of Smalltalk, the apparatus through which changes in the behavior of the "phenomenon model" can be reflected immediately in the graphics display and user requests to cause changes in the model can be indicated through graphic events.

The prototyping of this basic scientific repertoire in Smalltalk evolved in the following manner. The data (the object being visualized) plays the role of *model* to the display or drawing methods of a *view* which has scheduled a *controller* to handle interaction with the user AND with the rest of the system event management. Figure 1 shows the present classification. DataModel is the class that comprises the basic 2D graphing methods. (GKS compatible). DataView has the display methods supporting these 2D presentation methods. DataSurface is the class that comprises basic 3D graphing methods, inheriting from DataModel such graphing methods as are common to them both (e.g., normalizing data, scaling data and clipping to a viewport, calculation of tic values on an axis). DataSurfaceView (subclass of View) has the display methods supporting 3D presentation methods. DataContour comprises methods for the presentation of 3D data as two dimensional cross sections. The view supporting its display needs, ContourView, is a subclass of DataView. Each view class has a controller to manage its own *red button* and *yellow button* activities.

The current version of these tools uses a view design featuring a set of subviews serving as SelectionInList control panels and a subview serving as the graphics area. A class method *openWithPanelsOn* specifies the view layout and the names of the list collection and selection handlers that appear in and manage the panels. The list collection itself and the methods named in it are actually methods of the model rather than of the view. This makes it very easy to use the same view method with a variety of models, since each model can tailor the command panels to its own needs. Any object that wants some numbers graphed can declare an instance of DataModel , DataSurface or DataContour and make use of the desired graphing methods. Or, a computational model can be subclassed directly off

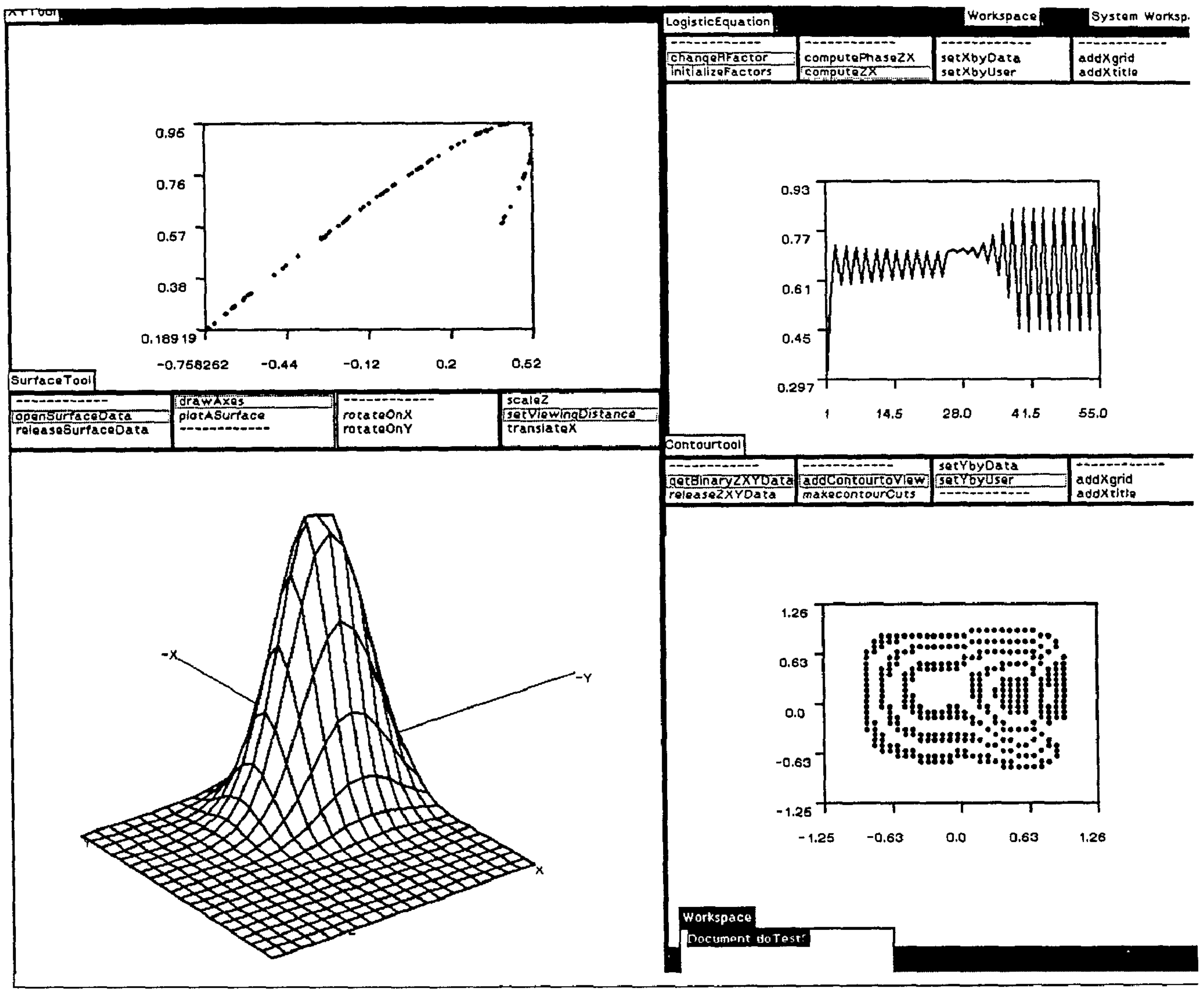

Figure 2: **Smalltalk screen showing graphics utilities. Clockwise from top: a Model (LogisticEquation) in xytool format; Contourtool; Surfacetool; xytool running a phase plot.**

one of these visualization classes so that the graphic behavior can be inherited or modified. In figure 1, LogisticEquation, Oscillator, and Flag are examples of such models. In addition, each of the above described views provides a class method to open an instance of itself as a standalone tool that can access and display precomputed data files. A spreadsheet version based on Pluggable Views allows a model or a collection of models to open a composite view in which each subview can be an instance of a different view class. Figure 2 shows a sampler of these tools. Figure 3 shows the spreadsheet opened on a model partitioned for parallel computation.

An earlier iteration subclassed each visualization object according to the type of storage convenient to its transmission as a data collection. In that version, graphic objects based on xy data were created as instances of class DataSet, itself a subclass of OrderedCollection while xyz data sets were created as instances of Matrix, a column/row collection behavior subclass of Object. Upon further development, it became evident that the manner in which data is organized for storage is not fundamental enough to graphic behavior to serve as the pivot of the inheritance mechanism. In the present iteration, the basic visualization class, *DataModel*, knows how to access and arrange (store) items in a variety of formats (e.g., unary indexable collections, matrices, etc.). This structuring makes it more convenient for a computation model to request a variety of graphic representations of itself.

2.2 Performance of the Scientific Interface

The Smalltalk environments in which this work has been proceeding are among the most powerful currently available. Specifically, the scientific classes

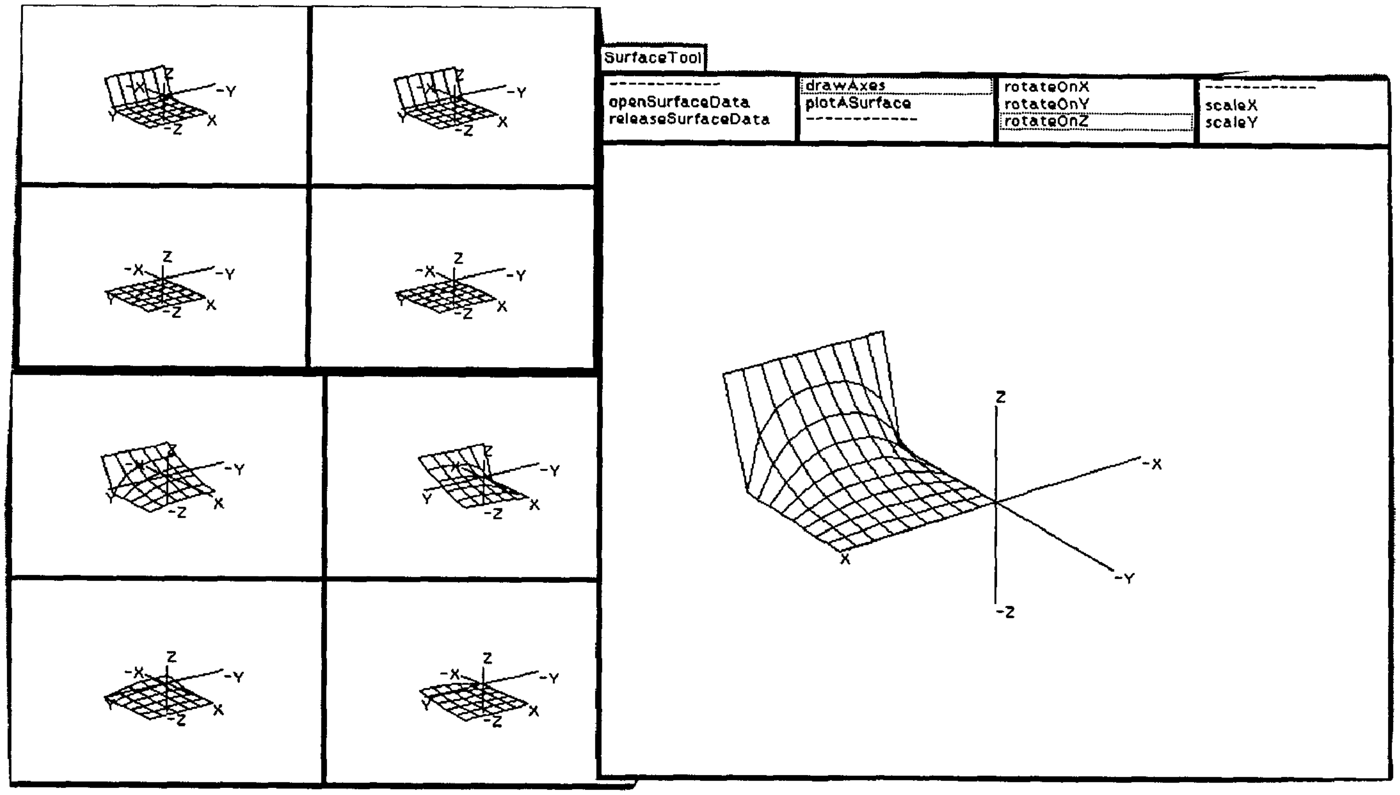

Figure 3: **Spreadsheet on a domain decomposition problem. Each cell is computing a region of the problem's physical space. Initial conditions are shown in the graph at top left. The final computation at each subdomain is shown at the bottom left. The solution as reconstituted from the cells can be viewed and manipulated in the tool shown at the right.**

have been implemented under ParcPlace version 2.3 on Sun3's† (8Mb memory), ParcPlace version 2.4 on a Sun4/110† and on an Ardent Titan† (32 Mb memory, dual processors), and Tektronix Smalltalk version TB2.3.0a on the Tektronix 4405†, 4406† (monochrome) and 4317† workstations (color, 12Mb memory). In this class of workstation, we have been able to achieve interactive processing speeds sufficient to show that Smalltalk is indeed a viable environment for prototyping bonafide scientific problems (not mere demonstration exercises) and that it can perform as well as postprocessing visualization graphics under SunViews or Xwindows.

Achieving effective performance has depended on maximizing the throughput of large amounts of floating point information, a throughput that involves storage, retrieval, and transmission as well as computation. Since we are aiming at a distributed computing environment, with the Smalltalk component serving as the *front end* to multiple processor facilities, it was essential to develop our strategies to reveal the areas of Smalltalk performance that would have to be enhanced. In the first instance, we wanted to see how these Smalltalk graphing tools would do as a post processing visualization environment on data generated by actual scientific prototyping tasks. A data set of 30,000 xy pairs (60,000 floating point numbers) or an 80x80 matrix of xyz values (19,200 float values) represents the upper bound of the 2D or 3D data volume expected from a prototyping exercise.

To understand the processing issues, consider a data file of 1000 floating point numbers. If the data is represented in ASCII format, each floating point value uses twelve to thirteen ASCII characters (depending on sign) to represent one single-precision number. In addition, a delimiter character (carriage return, line-feed, comma, etc.) is needed to separate the numbers from one another. A file of 1000 floats represented in ASCII normally consumes 12000 to 14000 bytes. In binary format, with each 32 bit float (or real *4) comprising 4 bytes, the file size is cut to 4000 bytes. In the former case, since the exact number of ASCII characters representing one float varies depending on sign, reading an ASCII stream requires a "grope until delimiter" type method based ultimately on a byte by byte inspection. The optimal way to introduce externally computed data is by accessing an entire binary file in one operation, attaching it to a ByteArray and parsing it directly into Floats.

Thus, one of the strategies important to data throughput in a scientific user's Smalltalk environment is the provision of methods to pack and unpack various collections of Floats or objects that include Floats, (e.g., Triplets, Vectors, and other scientific computation objects) into the barest possible byte sequences for communication between Smalltalk and nonSmalltalk environments. In fact, the *coexistence of a scientific element as a Smalltalk Object and as a byte sequence* is turning out to be the linchpin in the current development of a high performance scientific Smalltalk. The conceptual complexity and computational intensity of scientific prototyping has led us to distinguish between the exploratory and the validation stages of prototyping. The *exploratory* stage is Smalltalk's forte. Objects can be designed and discarded as they fail to pan out or retained as tentative commitments in a larger exploration. Performance speed is rarely an issue in this phase since preliminary models, or at least their components can be scaled down. However, a point occurs at which a set of primary Objects has proven to be a reasonable conceptual design but the dynamics of their behaviors must be drawn out with physically meaningful parameters and for an extensive duration. It is in this checking out stage that the computational intensity slows down the performance; yet the primary Objects need to remain interruptible and inspectable. For this reason, we are pursuing the development of scientific objects that carry two versions of some of their methods, a version entirely within Smalltalk, and a version written as a user primitive (linked C code).

3 Scientific Primitives

3.1 Visualization Primitives in the Model

Profiling the behavior of the 3D graphing methods quickly identified the places where execution in Smalltalk was more costly than the results warranted. Consider the treatment of a data set representing a surface as a mesh. Each xyz point is a Smalltalk object, namely, a Triplet (xyz point). The data set of Triplets has been organized as a Matrix, an object that knows how to store and retrieve its elements by rows and by columns. To transform this data so that it can be drawn as a three dimensional surface, each Triplet must be 1) fetched, 2) multiplied by a 4x4 Matrix of Floats, 3) further multiplied and divided by certain screen-based values to compute the 2D perspective point. These transformed points are 4) stored in another instance of Matrix where they can be 5a) fetched as an OrderedCollection of rows and columns for drawing as surface lines or 5b) as a SortedCollection of polygons, in depth order for representation as a shaded surface. So far, these methods are primarily computation and ordering methods of DataSurface. 6)Actual drawing of the 2D points is accomplished by DataSurfaceView.

As we have indicated elsewhere,[4] on a Tektronix 4317 running a Tektronix Smalltalk image, the processing time for a data set of 6400 xyz points (19,200 floating point values) transformed by a 4x4 matrix (steps 1 and 2 only) took 81657 milliseconds or about 1.2 minutes when done in Smalltalk with no user primitives (although the Smalltalk methods for computation on individual Floats are themselves executed as primitives). When a user primitive was added that operated on the entire Matrix passed as an Array of Floats to a C routine linked into the image as a user primitive, processing time for the entire transformation was cut to 1000 milliseconds (one second). Similar speedups occurred on the Ardent Titan (ParcPlace 2.4) where processing time on the same data was cut from 55810 milliseconds (steps 1 and 2) in Smalltalk with no user primitives to 560 milliseconds with a user primitive doing the transformation computations. The difference in performance of the two workstations is primarily attributable to the difference in processors.

Adding user primitives under either the Tektronix or ParcPlace images is quite straightforward; although the calling protocols are different, the implementations are similar enough that primitives developed in one image can be quickly ported to the other. In fact, the C code that does the actual computing is identical; only the format for passing arguments and the functions that mediate the Smalltalk and C addresses differ and they are easily plugged in as "boilerplate" once you know how to use them. In terms of strategy, there is an interesting difference; in the Tektronix version, user primitives can access the address space of the object as it exists in the image. Thus, an Array of Floats (Smalltalk object) serving as the instance of the object invoking the primitive method or as an argument to that primitive can be accessed in the C code by a service function that assigns the starting address of its data space to a C pointer to an array of floats. Items of type *float* can be taken from and put in this array. However, the array cannot be enlarged from within the primitive. Upon completion of the primitive and return to Smalltalk, any changes made to the contents of the array in C are accessible as the Floats in the Smalltalk Array.

In the ParcPlace interface, the user primitive passes Smalltalk object names as arguments but the service functions supplied by ParcPlace "copy" the Smalltalk objects, that is, make their content accessible in *additional space* declared in C. In this case, the size of the space allotted to the C item can be dynamically allocated according to the size of the object, which can be queried in the C routine. The primary operational difference between the two interfaces is the manner in which anything affected in the C routine is retrieved in Smalltalk. In the Tektronix approach, the primitive can pass in a number of objects as arguments; any that are modified in their *C persona* are thereby modified in their *Smalltalk persona* since the same address space is accessed directly in both modes. The primitive need return only a completion code. In the ParcPlace approach, any modification to a *copy* of a Smalltalk object has to be the argument of a return function (a set of "service' functions are provided) that structures the C element into the requested object which is to be returned in Smalltalk. To affect a number of different Smalltalk objects in one primitive and make use of those modifications in their persona as Smalltalk objects, one needs to pack them into one returnable object. For our purposes, an array of Floats representing a number of Float arrays and individual Float values was returned from the C primitive and unpacked into its respective Smalltalk persona on retrieval in Smalltalk.

In actual practice we tend to develop primitive versions of methods as we identify areas of an object's behavior where performance is bogging down the interactive prototyping. We "prototype the prototype" by subclassing off the object whose behavior we are trying to improve and substituting primitive versions of selected methods until we achieve acceptable performance. Sometimes, we find that one Smalltalk method maps directly to one primitive. In other cases, it appears best to absorb several Smalltalk methods into one primitive whose behavior is complex but functionally targeted nontheless. For the transformation procedure described above, the user primitive works as follows: The Matrix of Triplets (the xyz data) is unpacked in column order into an Array of Floats. The values of the 4x4 transformation matrix [6], having been determined by the user's interactive choices of viewpoint, rotations, view distance, etc., are unpacked in column order into another Array of Floats. The Smalltalk method *transformMatrix: xyzarray by: valuearray* is linked to the C function *transformMatrix(xyzarray, valuearray)* which calculates the transformed xyz values and replaces them in the "returnable" array. That primitive performs all computations necessary to return a set of screen coordinate points (i.e, steps 2 and 3).

We are not currently including functions to replace step 5b (a software depth sort of polygons) with a primitive but that could be included in this primitive if desired; or a separate primitive for that function alone could be fashioned. The performance trade offs between calling many limited function primitives or calling a few complex primitives rests on the amount of packing and unpacking of objects in Smalltalk between primitive calls. Experience has shown that the Smalltalk methods that are so handy "logically", (e.g, sending a message to a Matrix to break out a column, accessing the objects in the column, asking them (the Triplets) to deliver their x, y, or z value) are costly because of the number of steps required to service all the referencing. Normally, we try to identify behavioral units that have general utility and whose functions, once requested, do not require user intervention.

3.2 Visualization Primitives in the View

The transformation primitives we have been describing so far are primitives that replace computation methods in DataSurface. The methods that do the actual displaying are in DataSurfaceView. These methods use Pens and Forms (including HalfTones representing colors on the Tektronix 4317's) to produce the screen events we see. To the extent that scientific visualization performance demands capabilities beyond those possible using BitBlt as the only graphics primitive, primitives could be added to tap hardware capabilities of powerful graphics systems. For example, the Ardent Titan graphics workstation features 24 bit planes, hardware vector line drawing, hardware z buffering. Although it runs a basic ParcPlace 2.4 Smalltalk which does not make use of these capabilities, we have begun adding graphic primitives to that image. Preliminary results using a C routine that accesses the Titan lowest level graphics software to draw line segments as a method of PrimitivePen (a subclass of Pen), show performance improved by a factor of three for typical wireframe surfaces. (A 20 x 20 matrix, or forty lines of twenty points per line took 2400 milliseconds in Smalltalk on the Ardent, using Pen methods; using an Ardent "direct graphics" function, the same figure was drawn in 800 milliseconds. When direct buffer control, vector drawing and zbuffering (to support 3D rotation) are added as primitives, we anticipate performance in graphics on the Ardent under Smalltalk to be comparable to its performance under its native graphics mode.

Moreover, it does appear that user added graphics primitives can be integrated with the model-view-controller behavior so that *blue button* commands to move, resize, reframe, and collapse views produce the expected result. What is needed are primitives that directly cache the bits affected by the user primitive drawline. In principle, it should be possible to support the hiding, restoring, and overlapping of views drawn and/or colored by user added primitives.

4 Physical Model Representation

4.1 Flag Simulation in Smalltalk-80

The applicability of the Smalltalk-80 environment to the development of numerically-intensive models was tested on a flag simulation model. This problem was chosen because of its conceptual complexity and

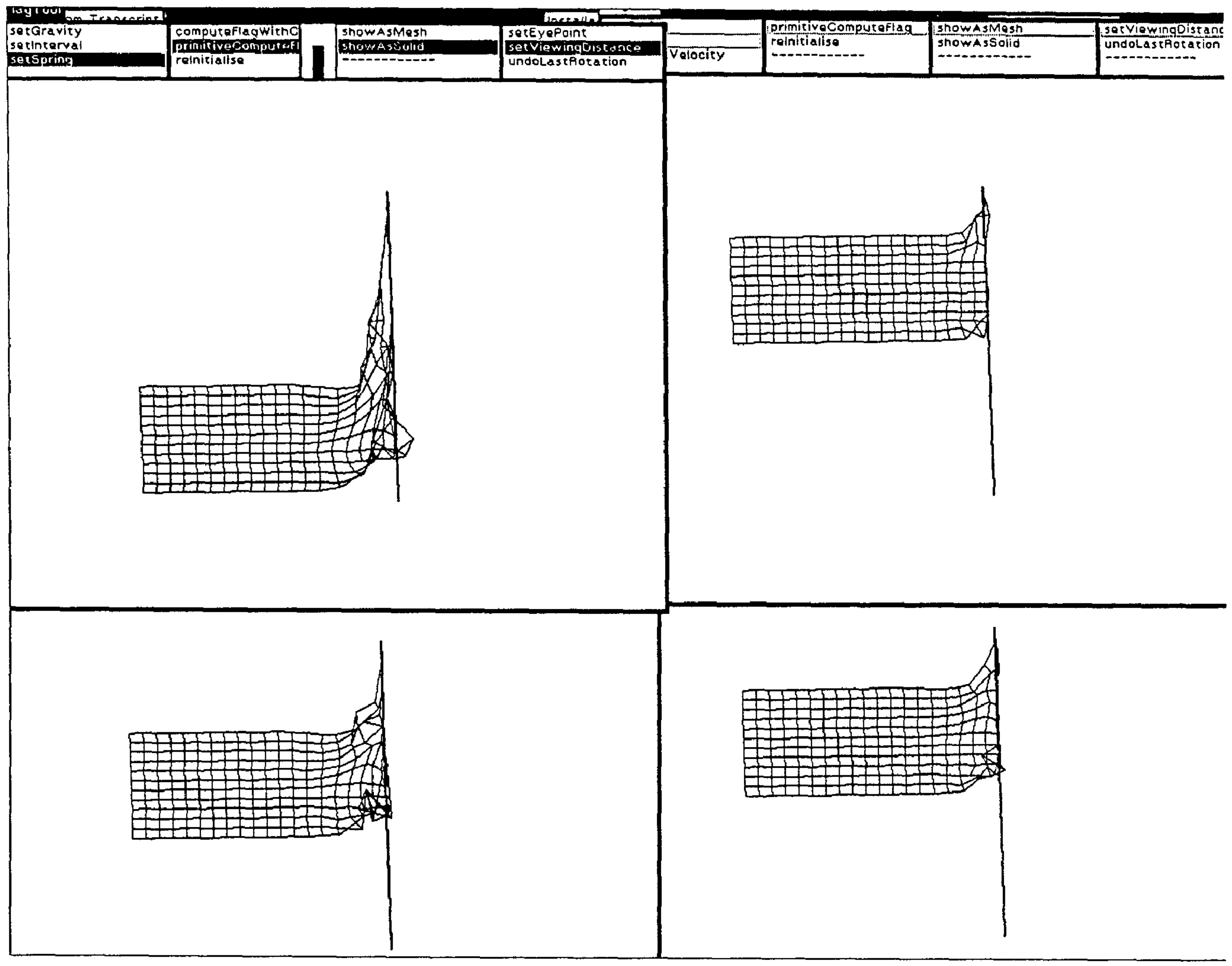

Figure 4: **Flag simulation. Clockwise from top right: 1) after 50 iterations; 2) after 80 iterations; 3) at 100 iterations; 4) 120 iterations at lapsed time of 83900 milliseconds on SUN4/110. From these results, the user was able to determine that the values used for gravity and for the spring force were too large since the model began to exhibit instability.**

as a means of testing the Smalltalk/computational-primitive paradigm. The flag model essentially consists of a grid of mass-points connected together by a set of springs and fixed in space at two points. The flag is subject to four forces in the model: spring, frictional, gravitational, and wind forces. The four forces on a mass point are given by vector relations. The flag "flaps" in three dimensions and, as a result, the bulk of the computation requires three-dimensional vector algebra in order to compute the force vectors. The computational loop for the model requires about 42,000 floating point operations for each time step for a flag of 12x20 mass points.

4.2 Computational Objects

The object-oriented model consists of the implementation of two new classes: Flag and Vector3D. As noted previously, Flag is a subclass of DataSurface and, thereby, inherits 3D visualization behavior. Instances of the Flag class are designed to respond to global messages pertaining to the forces and kinematical equations to which they are subject. Within the Flag instance methods, all three-dimensional vector operations are handled through the use of Vector3D objects, a subclass of Triplet. A three-dimensional vector expression, such as the computation of the spring force would be invoked as follows, given the normal, n, the velocity, v, and the wind velocity, vw:

```
springForce <- ( n * (( v - vw ) dot: n ))
```

As is evident, there is no need to deal with the vector components separately. The Smalltalk encapsulation of the vector methods into the class Vector3D allows the programmer to handle vectors simply *as* vectors. The vector methods presented form a very small subset of all possible vector methods. In spite of this, a great savings in "think-time" was achieved using this very simple set of methods. In fact, the prototyping of this physical model to the point where it exhibited physically correct behavior took about a week in Smalltalk.

In order to achieve a minimum level of physical realism, the flag model required a 12 by 20 grid of mass points. Each of these mass points was connected to its 8 nearest neighbors via 8 springs, each contributing a vector spring force contribution to the 'netForce' matrix, among other forces. In addition, the time differencing increment required to have stable flag behavior was such that 50 time iterations produced only a small visual change in the shape of the flag. The computational intensity of this model (with each iteration requiring some 42,000 floating point operations) provided a meaningful test for the feasibility of prototyping genuine scientific models in Smalltalk.

Again, performance was optimized by incorporating computational user primitives. The Vector3D methods to add, multiply, divide,etc. xyz points were implemented in C. In some cases, a computation loop (e.g., the calculation of the spring force at each point in the matrix was written as one primitive. Results for the flag model performance are given in Figure 5.

The incorporation of primitives led to an increase in debugging time, particularly in the latter stages of code development. While a very flexible, totally changeable environment is required at the start of a programming task, when the emphasis of debugging shifts towards visual interpretation of results, a faster-running version of the code is needed. One possibility for improving the primitive coding phase is the development of Smalltalk to C translation methods. Since many computationally intensive primitives will be programmed to run on parallel processing facilities, the issue of the *back end code* is a major topic on its own.

Finally, some future directions should be noted. Smalltalk's natural application to simulation often makes it easier to deal with modeling of physical phenomena than mathematical abstractions. When considering applications involving partial differential equations, Smalltalk classes to represent physical domain behavior can be readily constructed, often more easily than classes to effect formal numerical solutions. We are implementing a knowledge representation scheme [7] which uses physical domain behavior as its basis [4]. Also, use of knowledge representation allows us to incorporate very fundamental mathematical behavior into class protocols. This representation permits very significant speedup in numerical solution to differential equations by providing tight bounds. These bounds are very important for certain types of problems such as singular perturbation problems [9]. The increased performance provided by use of knowledge representation can do much to improve overall performance in a scientific Smalltalk system. That is, intelligent algorithmic techniques can reduce iteration requirements which can be a major source of performance degradation in Smalltalk.

5 Conclusions

The effort to date has convinced us that Smalltalk is potentially a desirable user interface for scientific numerical modeling. It offers a means to effect real computational steering [8], i.e. the *in-situ* modification of the model during computation. Our results indicate that a reasonable trade-off between primitives and Smalltalk code can be effected that will permit prototyping with credible performance. It may be that the balance between primitives and Smalltalk code will shift as the prototype develops. Clearly improved interpreter performance would be an important attribute because it would enhance the response of the Smalltalk code portions of the model. Special interpreter boards [12], [13] are one approach; faster processors now appearing in workstations are another. However, "information hiding" is the primary source of performance degradation in object-oriented systems. Work by Johnson [10] to effect inline code expansion offers promise of alleviating this problem.

As this work progresses, additional features must be added to any Smalltalk image if it is to support scientific prototyping. These include double precision for floating point, complex numbers (which have been added by Pinson[11]), additional graphics and visualization tools, and mathematical fonts with equation editors for problem input. In addition, the "back-end" processors will need some "standard" collection of numerical procedures that can be accessed as primitive methods. All of this is feasible, and some of it is work in progress. We remain confident that the interactive, object-oriented approach to scientific computation interfaces will evolve into viable alternatives to the traditional programming techniques, at least for model prototyping where the traditional methods restrict flexibility and productivity.

References

[1] Rice, J.R., SIAM News, Sept. 1987

[2] Peskin, R.L. and Russo, M.F. "An Object-Oriented System Environment for Partial Differential Equation Solution", 1988

Systems	*Smalltalk alone*	*Smalltalk with C Primitives*
ArdentTitan	3810 milliseconds	360 milliseconds
Tektronix 4317	7700 milliseconds	875 milliseconds

Figure 5: **Computation of one time step (42000 floating point operations)**

ASME Inter. Computers in Engineering Conf., San Francisco, Aug. 1988

[3] Russo, M.F., Peskin, R.L., and Kowalski, A.D., "A Prolog Based Expert System for Partial Differential Equation Modeling", *Simulation*, a publication of "The Society for Computer Simulation", San Diego, California, Volume 49, No.4, October 1987.

[4] Peskin, R.L., Walther, S.S, Froncioni, A.M., "Smalltalk - The Next Generation Scientific Computing Interface?", 1988 IMACS Conference on Expert Systems for Numerical Computing, Purdue University, LaFayette, Indiana, Dec. 1988 and to be published in the journal of Mathematics and Computers in Simulation.

[5] Walther,S., "Strategies for Interactive Graphing of Numeric Results", International Symposium on AI, Expert Systems and Languages in Modeling and Simulation (IMACS), Barcelona, Spain, June 1987, Proceedings.

[6] Newman, W., Sproull, R.,*Principles of Interactive Computer Graphics*, McGraw Hill, 1979, Chapter 22.

[7] Balaban, D., Greiman, Durst, M., and W., Garbarini, J. 'Knowledge Representations for the Automatic Generation of Numerical Simulators for PDEs". 1988 IMACS Conference on Expert Systems for Numerical Computing, Purdue University, LaFayette, Indiana, Dec. 1988 Proceedings.

[8] "Visualization in Scientific Computing", McCormick, B., DeFanti, T. , and Brown, M., NSF Report on Grant ASC-8712231, July, 1987

[9] Russo, M., "Automatic Generation of Parallel Programs using Nonlinear Singular Perturbation Theory", Doctoral Dissertation, Rutgers Univ., Jan. 1989

[10] Johnson, R., Graver, J.O., and Zurawski, L., "TS: An Optimizing Compiler for Smalltalk",OOPSLA Conference Proceedings, Sept. 1988

[11] Pinson, L.J., and Wiener, R.S., *An Introduction to Object-Oriented Programming and Smalltalk*, Addison-Wesley Pub. Co., 1988.

[12] *IEEE Micro*, February 1988, p.6-7

[13] Pountain, D., "Rekursiv: An Object-Oriented CPU",*Byte*, November 1988, p.340

*This research was supported in part by the Parallel Computing Laboratory of the Center for Computer Aids For Industrial Productivity(CAIP) and in part by National Science Foundation under grant number ECS-8814937. CAIP is supported by the New Jersey Commission on Science and Technology, Rutgers—the State University of NewJersey, and the CAIP Industrial Members.

6 Acknowledgements

We wish to thank Mike Miller of Tektronix, Inc. for making a preliminary version of his Cprim User Primitive package available to us. Andy Froncioni of Caip programmed the Flag Model.