The “No-Paradigm” Programming Paradigm for Information Visualization

E.H. Blake and H.A. Goosen

Computer Science, University of Cape Town
Rondebosch 7700, South Africa
E-mail: {edwin,goosen}@cs.uct.ac.za

Abstract We describe our exploratory visualization environment, the interactive Inventor Shell (iIsh), and the fact that no single programming paradigm underlies it. IIsh is an environment for interactive exploration of large databases of multidimensional abstract data, an application known as *Information Visualization*. This environment has been used in a number of areas but is still evolving — this flexibility is a key feature. IIsh is built around the Tcl scripting language and the Inventor[1] three-dimensional graphics toolkit, and simplifies the creation of interactive three-dimensional visualizations of abstract data. A particular feature of iIsh is that the interaction behavior of the system can be easily modified at run-time. We have used iIsh to construct a variety of visualization applications in fields ranging from computer architecture to medical insurance, and we describe our experience.

1 Introduction

The interactive Inventor Shell (iIsh) is an environment for Exploratory Multidimensional Abstract Data Visualization or *Information Visualization*. We have developed this environment for the interactive exploration of large databases of abstract information. We have used iIsh for visualizing shared-memory parallel program performance, manufacturing scheduling information, an astronomical database of galaxy information, medical insurance claim information, financial market data, and teletraffic data.

The design space of information visualization systems is even larger than that of Scientific Visualization systems because there are no physical objects to ground the concepts portrayed. The requirements of Information Visualization place a high premium on fast modification of systems, the maintenance of multiple versions reflecting different possible solutions to problems, and the ability to generate diverse high quality graphical interfaces.

IIsh uses interactive three-dimensional (3D) graphics techniques to display large amounts of data, enabling the human visual system to derive insights from the spatial, color, and texture information in the images. The use of three dimensions is important for two reasons. First, it enables us to display a much larger amount of data than would be possible in two dimensions. Second, it provides an extra dimension in which we can display interesting correlations. The interactive nature of the user interface complements the 3D aspect by allowing the user to focus in on interesting aspects of the data,

[1] Inventor is a trademark of Silicon Graphics Incorporated.

while ignoring irrelevant detail, further improving our ability to display large amounts of data.

In this paper we describe our visualization environment, and the programming paradigms that underlie it. A theme of the paper is that we are presenting a snapshot of a system and a field that is in rapid development. As such the emphasis of a system has to be on flexibility and extensibility rather than on structure and formalism. We are convinced that only once the subject matter is better understood can more supportive but restrictive methodologies (let alone "paradigms"!) be formulated. As a result of our experience we can present design criteria (Section 2.1) that in a presentation, such as this paper, belong before the discussion of the system but which were of course uncovered as we performed our experiments.

The organization of the paper is as follows. First there is a general discussion of the programming requirements of information visualization systems (Section 2). We then briefly discuss previous approaches to visualization programming in this context (Section 3). Our system, iIsh, is then introduced in two stages, firstly (Section 4) its design and salient features of the implementation, and then (Section 5) a few highlights of our experience in using iIsh. This experience is then discussed in Section 6 "No Paradigm or Multi-Paradigm?". This is followed by a few concluding remarks and pointers to further work.

2 The Programming Requirements of Information Visualization Systems

In its present state of development, the field of information visualization has no fixed methodologies, nor established problem areas with recognized solutions. This essential characteristic carries a number of implications when experimenting with visualization systems and developing visualization solutions. One has to be able to explore multiple solutions in parallel and to design new systems in an iterative design-implement-evaluate cycle. A further ramification is that there is no clear distinction between development-cum-experimentation systems and "production" systems. As a result there is also a need for the distinction between user and application programmer to be blurred.

In visualization, the importance of the user is always recognized. The whole aim of the visualization is to create an internal representation of the information in the mind of the user.

A visualization processing pipeline is presented in Figure 1. The first two stages, data modelling or acquisition followed by processing, are common to graphics pipelines. The need for flexibility means that the nature of the third, display, stage is not fixed and that it can be easily changed. Various forms and degrees of preprocessing can be applied to the data. There isn't any notion of graphical "realism", instead the ideal of truth in representing the underlying data is maintained. It is acknowledged that prettiness may stand in the way of insight. The addition of the final step, that is, decoding by the user, is an emphatic difference with standard graphics pipelines. The role of interaction is emphasized by the feedback loop.

There has been surprisingly little published on systems that allow three-dimensional graphics and interactive exploration techniques to be employed for information visual-

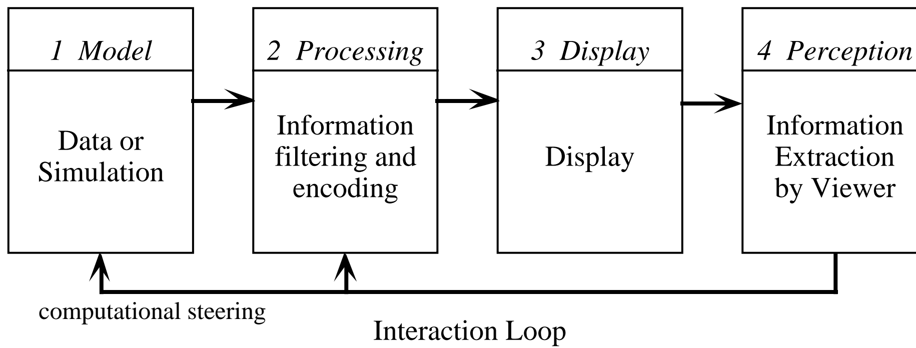

Fig. 1: A visualization pipeline.
The role of the viewer and computational steering is indicated in this diagram.

ization. Partly this is due to a lack of suitable tools. The scientific visualization packages like AVS, Explorer, etc, have poor support for more abstract information, focusing instead on continuous fields and support for more traditional scientific and engineering representations. Also, scientific visualization packages typically do not provide good support for interaction and particularly new forms of interaction. In our experience, it is difficult to extend these packages to include the required features for information visualization.

2.1 Key Design Criteria and Concepts

During a process of implementation, testing, and progressive refinement of ideas, we developed the following set of criteria (which are not independent) to support the design:

Immersive manipulation — the user interacts directly in 3D (by using a pointing device) with an immediate graphical representation of the data or data attributes, without the intervention of agents like 2D controls (scrollbars and other traditional direct manipulation tools), or 3D widgets that are separate from the data representation.

This is an extension of the original formulation of direct manipulation [12], and is in contrast with interfaces based on 3D widgets [16]. The VIEW system supports a similar notion of direct interaction with the image [2].

Input-output coupling — browsing of the display (output) image, and then using it as an input tool for the refinement of queries. The output of the browsing operation can be used as the input of another browser on a different set of attributes, and so on to any level of nesting [1].

Flexibility, power, and extensibility — these are not directly supported by the interface of the system, instead being provided by the underlying programming constructs. These are certainly vital to the success of a visualization system, but we consider that current attempts to provide these in the interface have not succeeded. The

users of visualization systems are not professional programmers though they are competent and sophisticated professionals. These users combine needs in terms of ease of use and protection from their mistakes as novice programmers with the needs of flexibility, power and extensibility.

Dynamic embedded spreadsheet — the spreadsheet model of user-level programming has been proven to be accessible to general users (especially in the business community), and offers a concise and powerful and comprehensible programming model. While much of user-programmable system design makes use of visual programming techniques (e.g., the data-flow diagrams of AVS, etc.), the spreadsheet model and dataflow visual models are functionally equivalent.

Focus — related to browsing and refinement of queries. Focusing (or filtering) helps us to get rid of meaningless clutter, by reducing the amount of data displayed.

Correlative linking — techniques to make it easy to create correlation views. Interactive visual correlation allows one to have different displays relating to the same information. Shneiderman calls this tight coupling, referring to the extension of direct manipulation.

Generality — provided by the ease of modifying the specialized system. An analogy is a shifting spanner, which is specialized for one nut, but can easily be changed to accommodate different sizes.

Computational steering — while visualization systems have always recognized the importance of the user as *viewer* they have been less accommodating of the user as *interactor*. A user interaction loop which goes back to the model is given a special name, “computational steering”.

Standard building blocks — we created our environment for interactive exploration by combining two widely available tools: Tcl, an extensible scripting language, and Inventor (based on OpenGL, which is fast becoming a standard for interactive 3D graphics). The environment is familiar to many users, powerful, and designed to be easily extensible. Because it is based on Tcl, iIsh can easily be incorporated as a building-block into other Tcl-based tools.

Finally, when these criteria, and particularly the central importance of interaction tools, are taken into account, then the standard visualization pipeline presented above, for all its novelty, becomes too restrictive. It seems better to regard the various aspects of a visualization system not as stages in a pipeline but as a collection of cooperating processes that act on a common pool of data ([15]). Once more the benefit of this is to emphasize flexibility. Figure 2 shows the various stages of the visualization pipeline arranged round a common cloud of dynamically changing data. The viewer is dependent on display and input devices for interaction with the data.

3 Previous Approaches to Visualization Programming

In this section we discuss the basic enabling technologies that contributed to our iIsh system.

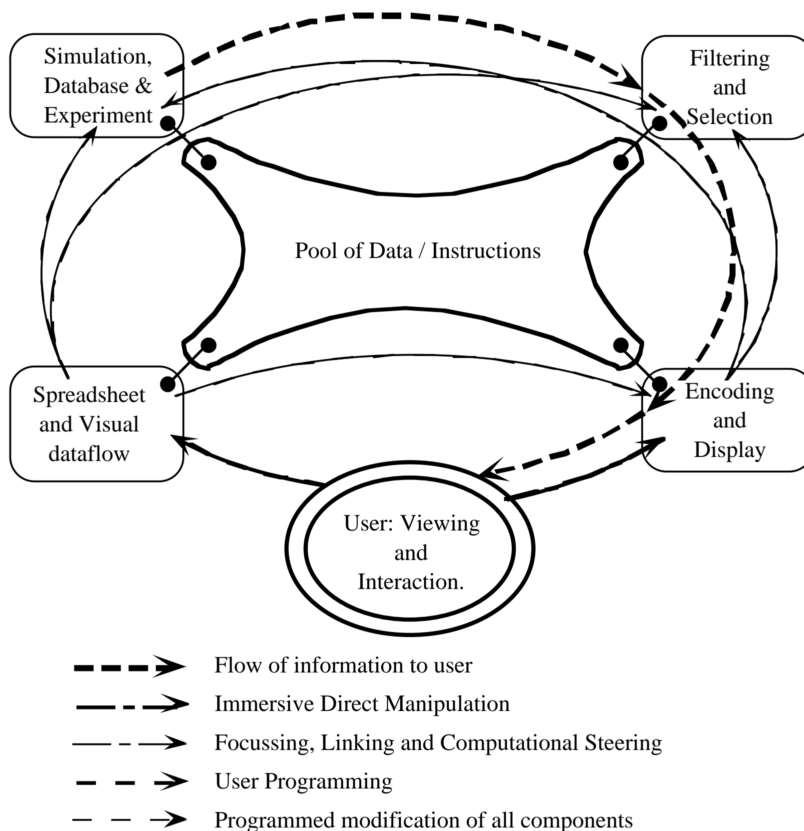

Fig. 2: A Conceptual Model of a Visualization System.
Visualization systems allow the user to view complex data. They also allow sophisticated interaction with many aspects of the data and the models. The notion of plugable processing, output and input modules should be contrasted with that of a pipeline with a feedback loop.

The Contribution of Graphics Programming Methodologies

The Eurographics workshop series on Object-Oriented Graphics has reflected a certain 'eminence' of object-orientation in interactive systems, graphics, and computer animation [3, 9]. The *Doré* system, which was already well established by the time it was presented at the first workshop, was a major first success of object-oriented graphics.

The iIsh system is (by its very name — interactive Inventor Shell) based on the Inventor C++ toolkit from Silicon Graphics [14]. Inventor is an object-oriented toolkit that supports the programming of direct manipulation interfaces in the immersive sense — allowing the user to interact directly with the displayed 3D objects in the same window as the display and without having to seek recourse to manipulation widgets.

The Inventor library provides a collection of retained high-level 3D objects and

makes them available for building interactive interfaces. The contrast is with earlier interactive graphics systems where rendering of 3D objects was the prime focus and "picking" of objects was the most sophisticated user input action possible.

An Inventor 3D scene and associated interactions are stored as a direct acyclic graph of nodes (it is analogous to traditional display lists). There are *shape nodes* for the geometry, *property nodes* for attributes, and other nodes for *interaction* to act on events.

The philosophy of Inventor is that flexibility is the overriding concern. Questions of efficiency are important but secondary, while structure and policy enforcement are considered least important. Inventor is object oriented, but with many extensions. The 3D scene graph is a mixture of aggregation relations that build up 3D objects, attribute inheritance trees, procedural rendering instructions[2] and asynchronous event handling code.

Objects in Inventor come in two guises: *nodes* which are open objects storing state, and objects representing single *actions*, with possibly some associated state information. This extension of object-orientation provides the user with the ability to design new actions (methods) for existing system classes without having to create subclasses of the existing classes. An action can be applied to objects in a scene graph and it then processes the nodes in the graph. The state held by the action object provides the inheritance and accumulation of attributes during traversal of the scene graph.

Inventor also introduces *node kits*. The previous paragraph described how Inventor extends object-orientation to provide methods divorced from classes (the *actions*) and opens up the internal state of an object to allow various useful features. The node kit should be seen then as a re-introduction of many of the traditional encapsulation and enclosed methods associated with object-oriented programming. The node kit provides a collection of subgraph nodes that "relate to 'objects' (chairs, bicycles) in the 3D scene" [14, p. 346]. Thus a node kit in many ways reassumes the role associated with classes in object-oriented programming.

Scripting Languages

Tcl (Tool Command Language) is a popular and flexible scripting language [11]. Tk extends Tcl to support the writing of graphical user interfaces. Languages like Tcl are now frequently used in visualization systems (for example, VIEW mentioned earlier uses a language modelled on Smalltalk-80 [2]). In accordance with our desire for flexibility, we have opted for one of the least structured of these languages, believing that in the absence of clear guidance on what our user will require we want to start off providing the maximum flexibility.

Tcl is interpreted and the interpreter can be *embedded* in the application. Tcl can be regarded as providing a powerful command interface to the application. Alternatively the application can be regarded as extending the Tcl language with application specific semantics. These views are equivalent. The power of the conceptual model is further that the user's Unix shell interface is extended (since Tcl provides the capabilities of a shell) to become the interface to the application. Tk adds window management and event handling to this powerful interface. Finally, the Tcl/Tk user communities is expanding rapidly and providing further enhancements to the toolkits.

[2] The nature of a displayable object depends on its parents and preceding siblings.

The limitations of Tcl are an excess of its strength: flexibility. Tcl provides only two data types: strings, and associative arrays. There are also built-in commands to manipulate lists. Multidimensional arrays have to be simulated, and arithmetic is inefficient because numbers have to be converted between string and internal representations.

Spreadsheets

Our application building environment incorporates a spreadsheet module to augment the standard data-flow models used in systems like AVS and Iris Explorer. Our spreadsheet is embedded in a dataflow environment, with the dataflow relationships specifying the inputs and the outputs to the spreadsheet. Unlike conventional spreadsheets that are loaded up with static data, our spreadsheet is part of a processing pipeline, and dynamically fires when new data becomes available.

Spreadsheets have made their appearance in computer graphics [10, 8]. Levoy's system uses spreadsheets where the individual cells contain graphics, and the formulae are programmed with Tcl [10]. He cites the limitations of dataflow visualization system in terms of expressive power and scalability to motivate the use of spreadsheets. We have chosen to separate the spreadsheet interface model from the graphical direct manipulation style of interaction.

Hudson discusses the advantages of spreadsheets as user interfaces and points out that spreadsheets embody a kind of dataflow computation [8, p211]. One might prefer to say that both dataflow and spreadsheets embody many aspects of the declarative or functional programming paradigm.

Spreadsheets have a distinct declarative feel, in the sense that the user specifies relationships that have to be maintained between cells. In addition, a spreadsheet spreads the problem out in space, in a declarative fashion, rather than in time sequence (or procedural fashion).

Dataflow derives meaning purely from geometrical relationships. If the graph gets complex, the user gets lost. Spreadsheets retain the declarative feel and the spatial localization, but adds naming, to control the visual complexity.

4 Design and Implementation of iIsh

IIsh is an interpretive environment for creating interactive 3D visualizations of abstract data. It includes a scripting language, a visual programming system, and 3D viewers that support immersive manipulation.

A user interacts with iIsh in one of three ways. First, iIsh provides a window into a three-dimensional space populated by 3D view objects. Zooming, panning, camera rotation, camera translation, and selection can all be done using only a mouse within the 3D window, and without requiring 2D or 3D widgets as intermediaries. Selection can be either on view objects as a whole, or else on components of view objects, for example, on an individual face or vertex.

Second, a two-dimensional drawing editor, called Vizion, is used to specify which view objects should be drawn, and what the relationship between the input database and each view object is. Vizion supports a visual programming system based on dataflow

diagrams (much like that used in Iris Explorer or AVS), using functional blocks interconnected by lines indicating the dataflow relationships. However, the dataflow diagram is *not* intended to be the primary way of programming the visualization. Instead, Vizion provides an *embedded spreadsheet* functional block that accepts arbitrary iIsh scripts as formulas. Unlike more conventional spreadsheets that have cells containing static values, the embedded spreadsheet module has input and ports that are connected to data sources. The input ports feed into cells, and cells can contain lists of values. Cells can also be connected to output ports, supplying data to the rest of the system, in particular to the view modules.

Third, an interpretive shell allows the user to program the system by entering iIsh scripting language commands. This general programming interface supports features that we have not yet decided how to provide in the Vizion programming system, for example, the binding of event handling scripts to events. Figure 3 shows the iIsh 3D viewer (displaying a cone) and the Vizion visual programming window with a spreadsheet module.

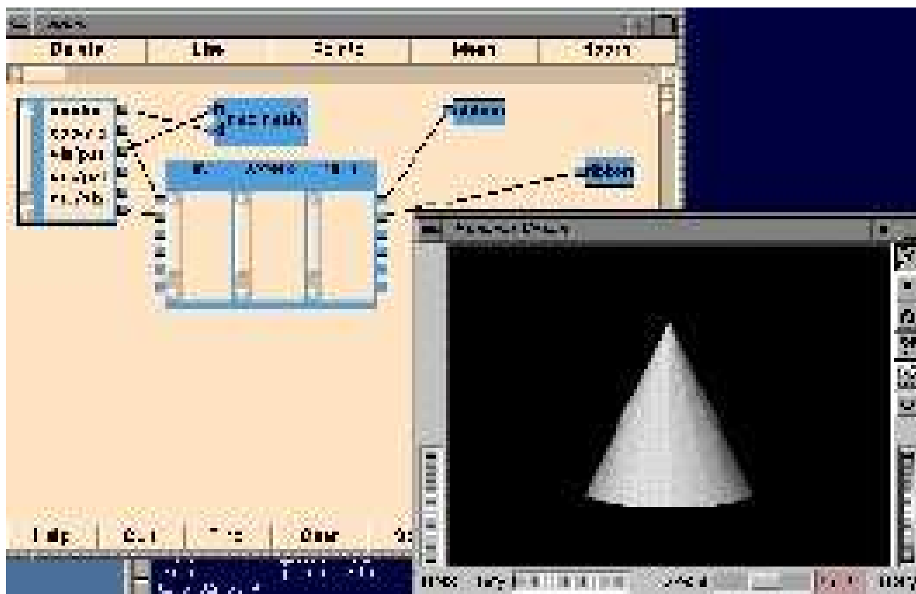

Fig. 3: iIsh windows.
The iIsh 3D window displaying a cone, and the Vizion visual programming editor with a spreadsheet module.

IIsh achieves its functionality by creating and manipulating Inventor scene graphs through an interpreted scripting language. It supports an event binding mechanism that allows iIsh scripts to be dynamically bound to scene graph components.

Compared to more traditional programming tools based on compiled programming languages and libraries, the iIsh scripting language provides

- a short write and test cycle,

- an easy learning curve, and
- a convenient and compact way of storing different versions of a visualization program.

4.1 Design

The iIsh scene graph structure is considerably simpler than the Inventor scene graph structure. As in Inventor, a scene graph consist of nodes that are connected to form a directed acyclic graph. There are two kinds of nodes: shape nodes, and group nodes. The shape nodes represent simple graphical shapes (cubes, spheres, cylinders, or cones) or more complex shapes (meshes, text, or nurbs surfaces). Because iIsh nodes are based on Inventor node kits (mentioned above), the material properties, coordinate transformations, and other information required to render the shapes are all contained within the iIsh node itself, and we do not require property or transformation nodes as in Inventor.

An iIsh scene graph can be active or inactive. When a scene graph is activated, it will be rendered, and the active graph also automatically supports the dynamic binding of iIsh scripts to scene elements.

There are five basic iIsh commands to enable node creation, scene graph composition, setting of graphical attributes (for example material properties of shapes), the binding of iIsh scripts to graphical components, and the activation of scene graphs.

For example, the iIsh command line

```
iCreate Cube my_cube
```

consist of three strings. The first string, `iCreate`, is the name of the iIsh command to create a node. The second string, `Cube`, tells iIsh what kind of node to create (in this case a cube). The third string, `my_cube`, is the name we give to the node, so that we can refer to the cube node later in the program.

The various types of iIsh commands can be summarized as follows:

- `iCreate node_type node_name` — `node_type` specifies which node to create, and `node_name` associates a name with the node.
- `iAddChild parent_name child_name` — `parent_name` is the name of the parent node, and `child_name` is the name of the child node that will be attached to the parent node.
- `iSet node_name attribute_string` — The `attribute_string` will be passed to the node identified by `node_name`. The underlying Inventor node kit mechanism supports the use of such strings to set the values of parts contained in the node kit.
- `iBind node_name event_specifier script_string` — `node_name` identifies the graphic element[3] we want to interact with, `event_specifier` specifies an event, and `script_string` is a string which will be interpreted by

[3] Note that this is not the same as a node, because a single node can represent many graphic elements, uniquely identified by giving a path name to the node representing the graphic element.

iIsh when the specified event occurs. The `script_string` can contain special character combinations that are replaced by event-specific information before the script is interpreted, as described below.

- `iSetSceneGraph node_name` — `node_name` identifies the node that should be considered the root of the active scene graph.

For example, here is a complete iIsh program that creates a green cube named `my_cube`, a red cone named `my_cone`, and draws the cone 2 units away from the cube. When the left button is pressed on the cube, "mouse button pressed on green cube" will be printed on the standard output, when the left mouse button is pressed on the cone, "mouse button pressed on red cone" will be printed on the standard output, and when the left button is pressed anywhere else in the display area, "mouse button pressed on nothing" will be printed on the standard output.

```
iCreate Cube my_cube
iCreate Cone my_cone
iSet my_cube "material {diffuseColor 0 1 0}"
iSet my_cone "material {diffuseColor 1 0 0} \
        transform {translation 2 0 0}"
iBind .my_cube down1 {puts "mouse button pressed on green cube"}
iBind .my_cube.my_cone down1 {puts "mouse button pressed on red cone"}
iBind . down1 {puts "mouse button pressed on nothing"}
iAddChild my_cube my_cone
iSetSceneGraph my_cube
```

4.2 Implementation

IIsh is an extension of Tcl to support interactive three-dimensional graphics, in much the same way that Tk extends Tcl to support the writing of graphical user interfaces. The implementation of iIsh borrows heavily from the Tk implementation. The three-dimensional graphics primitives used in iIsh is supplied by the Silicon Graphics Inventor library (see above). IIsh supports the standard Inventor file formats for input and output of scene graphs.

The iIsh implementation is based on an interpretive interface to the Inventor node kit mechanism. Node kits allow text strings to be used to set attribute values, greatly simplifying the interpretive interface. For example, iIsh does not have to support all the member functions used to set the values of fields in Inventor node classes. Instead, a single command (`iSet`) is used to pass text strings to the node kit `set` member function.

Tcl (Tool command language) is an extensible interpreter developed at the University of California, Berkeley. It is freely available, and many extensions are have been developed to support application requirements like graphical user interfaces (Tk), distributed programming (Tcl-DP), and database access (Tcl-SQL). Tcl provides basic facilities like variables, associative arrays, string processing, and procedure calls.

IIsh is a simple extension to Tcl/Tk. All the Tcl/Tk commands are available in iIsh. IIsh node names reside in a separate name space from Tcl variables. IIsh supports an explicitly controlled stack of name spaces, which helps to keep name space

pollution under control in procedure calls, but without requiring modification to the Tcl interpreter. The iIsh name space stack is implemented using a linked list of hash tables.

The binding of iIsh scripts to graphical components is based strongly on the Tk binding mechanism. An arbitrary iIsh script can be bound to a specific event on a graphical element. A number of special replacement character sequences can be included in the script: appropriate values based on event information will be substituted for these characters before the script is executed. For example, the character sequence "`%w`" will be replaced by the window position, in pixels, of the mouse pointer when the event occurred.

Graphical elements are specified by composing a path name to the graphical element. Each path name element is the name of the node that has to be traversed to get to the shape node representing that element. The node names are separated by "." characters. The path name "." will match an event occurring anywhere in the display area.

The event binding mechanism is implemented using a hash table that stores *(path, event, script)* tuples. When an event occurs, a path name is generated for the graphical element associated with the event. For example, if the event is a left mouse button down event that occurs on the red cone in the above example, the path `.my_cube.my_cone` would be generated. The hash table is then scanned for tuples matching the path and the event. If no tuple is found, the trailing path element (`.my_cone`) is removed, and the hash table scanned again for a match. The search for a handler continues until the "." path has been checked. If a matching handler is found, the script associated with that handler is scanned for substitutions, and then the script is interpreted by the Tcl interpreter.

In addition to the iIsh extensions to the Tcl interpreter, we have also implemented a collection of C++ classes to encapsulate view objects that we have found useful. These classes allow the concise specification of a view object, based on the quantitative data we want to display. For example, our `VoMesh` object takes a list of name-value pairs, and creates a square mesh with the heights proportional to the values, and the names available for use in callback functions that support correlative linking to other view objects.

The three-dimensional viewing area in iIsh is provided by the Inventor SceneViewer, that already provides camera movement, rotation, and zooming features. The Vizion drawing editor to support visual programming is implemented entirely in Tcl/Tk, with iIsh providing the three-dimensional graphics primitives.

5 Experience with Using iIsh

IIsh was developed in an experimental environment, in the course of building several information visualization systems. These are presented below, in chronological order, to present our experience and reasons for particular design decisions.

5.1 Chiron Parallel Program Visualization

The original application we were interested in was the analysis of traces of parallel architecture behavior. We had constructed an execution-driven simulator that provides detailed information about parallel architecture behavior (especially cache memory system behavior) when executing programs, and visualization seemed a powerful tool to help us understand the behavior, given the large volumes of data generated by a simulator. Our initial experience with scientific visualization tools convinced us that we needed something more flexible, and we started implementing our visualization system, Chiron [7, 6] using Inventor, a C++ toolkit for implementing interactive 3D graphics programs.

Chiron displays several views of performance data associated with lines of source code and objects in the parallel program. Our primary focus was on providing fine-grain information about the behavior of individual objects, source lines, and cache lines in the program. The information included both temporal event information, and also summary information accumulated over selected time periods.

We designed several interactive views to represent this information. Each view is a 3D shape that can be individually manipulated. We also support the linking together of different views, so that the mouse selection of a particular point on one view can be correlatively linked to the graphical representations presented in the other views.

We have evaluated Chiron using several parallel applications [7, 6]. Although space does not permit a complete discussion of Chiron features, Figure 4 (in the colour plate appendix) show an example of Chiron views.

In the course of the Chiron development, we needed to maintain multiple versions of the program to experiment with. Apart from the cumbersome implement-compile cycle, we found several other problems:

- Object and executable files are large and consume significant amounts of disk space.
- It was difficult to maintain multiple versions of source files, synchronized to the executables, and to make sure that arbitrary features could be combined in one compilable executable.
- To keep track of the features in a particular version of the executable creates a significant documentation problem.

These problems motivated us to look for an alternative that would retain the flexibility of our C++/Inventor system, but that would eliminate these significant obstacles to experimentation. An extensible interpretive environment seemed like a natural solution. Tcl/Tk was chosen because it was fast, well-implemented, well-documented, and had a large user community.

5.2 Manufacturing

We developed a visualization system to assist the master scheduler in a manufacturing plant. The plant is run using a management information system implementing the MRP-II (materials requirements planning) methodology. The visualization system was

implemented in six weeks by two COBOL programmers who had no previous experience of computer graphics or C programming. The application required a custom COBOL interface to extract data from a database of manufacturing information. This data was then forwarded to our visualization module.

Two visualization modules supported the MRP-II functions of rough-cut capacity planning, and capacity requirements planning. A third module represented sales history data. Figure 5 (in the colour plate appendix) shows the rough-cut capacity planning module. The ribbons represent the available capacity, measured in machine hours, of each work center on a weekly basis. For each week, a colored cube indicates the total capacity required of that work center. The vertical position of the cube indicates the required capacity relative to the available capacity. In addition to the position, the cube is colored red if the required capacity exceeds the available capacity, and green otherwise.

To get more information, the user can click on a particular cube. That brings up a display of all the orders that were scheduled on the work center for the week in question.

5.3 Other Applications

Several other applications have either been completed or are under construction, and we briefly discuss these here. The Astronomy Department at the University of Cape Town does research on the large-scale structure of the universe. A database of all known galaxies in the Southern Sky has been compiled, and we implemented a visualization system for displaying this database. The system has been used to discover new types of large-scale structures [5, 4], and also to make video and slide shows for use in planetaria. Although this application does not require much interaction, it demonstrates the flexibility of our environment.

A second application we are working on is visualizing medical insurance data. Medical insurance companies collect large amounts of information relating to claims. This information is potentially valuable for designing fee structures, and also for discovering fraudulent use of benefits. However, there are few suitable tools available for exploring such databases, with the result that often the data is not fully exploited. The dataflow programming interface to iIsh allows the user to rapidly switch between different datasets, to view multiple datasets at the same time, or to link features on one data set to the representation of another dataset.

We are also exploring the visualization of financial market data. This is a demanding application area that could make good use of animation to show the real-time behavior of financial instruments. iIsh does not currently support the easy specification of animation behavior, and we are therefore working on animation extensions.

6 No Paradigm or Multi-Paradigm?

How did object-orientation fare under iIsh? As we have seen iIsh uses Iris Inventor as its underlying interactive graphics system. In general we found that *node kits* were needed to prevent our users from being overwhelmed by the flexibility provided by Inventor. Therefore to the extent that node kits mark a return to class inheritance and

encapsulation this represents a step back towards object-oriented orthodoxy from the potential for freedom from structuring inherent in Inventor.

As we have pointed out, both spreadsheets and visual dataflow provide essentially functional programming facilities. We have extended this with a full procedural language. However unlike Levoy [10] we do contain the imperative programming within the spreadsheet cells (in the same way that visual dataflow allows state in the individual modules), spreadsheet cells are not allowed to have side-effects on other spread-sheet cells. This greatly aid the understanding of users and controls some of the complexity of the spreadsheet.

6.1 Interaction

We believe that support for the flexible specification of interactive behavior is essential in an information visualization system. In information visualization applications, the relationship between data items are often non-linear. The dimensionality of the data is often much greater than in typical scientific data sets, increasing the need for examining many different types of correlation between data items. Our experience with trying to use traditional scientific visualization systems for information visualization leads us to conclude that more support for interaction is needed. Often the types of interaction that the user may require cannot realistically be anticipated by the designer of the visualization system, arguing for a programming interface to interaction specification.

The most basic kind of interaction is to query a visual representation, to find out more about the underlying data relating to a specific part of the image. For example, consider the visualization representing work centers in the manufacturing plant showed in Figure 5. The user of such a system may notice a particular week in which the needed capacity is particularly high, and would like to see, for example, detailed information on the order which is responsible for the high demand that week . This data is stored in a database, and to get the information, the system has to query this database, based on a user action (like mouse selection).

6.2 Interactive Correlation

A more complex interaction that we have found useful in our work is interactive correlation. For an example, refer to Figure 4, showing two surfaces. The surface on the left represents the execution time of individual lines of code in a program, while the surface on the right represents the execution time spent waiting for individual objects to be accessed in memory. To optimize the performance of the program, it may be useful to know which objects are accessed by which lines of code in the program. By selecting a source code line on the left surface (using the mouse), we can highlight all objects accessed by that line of source code on the right surface. In the same way, by selecting an object on the right surface, we can highlight all lines of source code that access that object on the left surface. The user is in effect browsing a database of correlation information, getting rapid visual feedback on the relative cost of different selections. These operations require access to a database of correlation information, and then modifying the images to show the correlation information.

7 Concluding Remarks

We believe it is too early for a programming paradigm (i.e., set of abstractions) for Information Visualization systems. Instead, we opted for a relatively unstructured collection of concepts, each appropriate to a particular need that we perceived. A summary of these needs and concepts are:

Exploration — Supported by immersive manipulation and correlative linking to browse the data space, and dataflow connections to select particular data sources and appropriate view objects.

Filtering — We provide an embedded spreadsheet module to program general filtering and data processing. The spreadsheet formulas are written in the Tcl scripting language.

Interaction — By binding Tcl/iIsh scripts to events and view objects, the user can change the interaction behavior to suit the particular needs of a data set.

Our initial experience with iIsh has been favorable, and we are continuing our experimental evaluation by building more information visualization systems, and getting people to use them. We are also extending iIsh as we discover new requirements that challenge the existing implementation.

Many issues are still unclear. First, it is difficult to find the proper balance between the use of 2D and 3D graphics for visualization. There are many cases where the use of 3D allows the display of more information, and where the position of objects in space can give clues about the underlying process one is analysing. However, it is also often the case that 3D confuses the user by presenting too much data that becomes simply confusing, or that a particular 3D display is difficult to interpret because background parts of the image is obscured by foreground features.

Another problem is the choice of an appropriate display method before one has seen the data displayed, as often happens in an exploratory environment. For example, a particular data set may be too large to display in a given format, given the limited performance of a particular machine. This can be frustrating in a supposedly interactive, exploratory environment. We are therefore examining features to allow iIsh to remain responsive and controllable while data is being processed, so that commands can be aborted or modified as the user receives incremental information.

Our spreadsheet module (in common with most existing spreadsheets) has a shortcoming in that it only supports a global namespace and therefore is not suitable for implementing large problems. We are investigating the use of a hierarchical spreadsheet model to address this.

Acknowledgements

This work was supported in part by the Foundation for Research Development (South Africa) and the University Research Committee of the University of Cape Town, South Africa. We would like to thank Dieter Polzin, Wayne Paverd, Matthew Shaer, Grant

Kauffman, Philip Machanick, and Peter Hinz for implementing and using various parts of Chiron, iIsh, and Vizion. Explorer and Inventor are trademarks of Silicon Graphics Inc., Mountain View, California.

References

[1] C. Ahlberg and B. Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In B. Adelson, S. Dumais, and J. Olson, editors, *Proceedings of CHI*, pages 313–317. ACM, 1994. (Boston, MA, USA, April 24-28, 1994).

[2] L. D. Bergman, J. S. Richardson, D. C. Richardson, and J. Frederick P. Brooks. VIEW — an exploratory molecular visualization system with user-definable interaction sequences. In SIGGRAPH '93 [13], pages 117–126.

[3] E. H. Blake and P. Wißkirchen, editors. *Advances in Object-Oriented Graphics, I*. Springer-Verlag. Berlin, 1991.

[4] A. P. Fairall and W. R. Paverd. Large-scale structure in the southern sky to 0.1c. In *35th Herstmonceux Conference on Wide-field Spectroscopy and the Distant Universe*. World Scientific Publishing Company, 1994. In press.

[5] A. P. Fairall, W. R. Paverd, and R. Ashley. Visualization of nearby large-scale structures. In C. Balkowski and R. Kraan-Koorteweg, editors, *Unveiling Large-Scale Structures Behind the Milky Way*. Astronomical Society of the Pacific Conference Series, October 1994.

[6] H. A. Goosen, P. Hinz, and D. W. Polzin. Experience using the chiron parallel program performance visualization system. 1995. Submitted for publication.

[7] H. A. Goosen, A. R. Karlin, D. R. Cheriton, and D. W. Polzin. The chiron parallel program performance visualization system. *Computer-Aided Design*, 26(12):899–906, December 1994.

[8] S. E. Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transactions on Graphics*, 13(3):209–239, 1994.

[9] C. Laffra, E. Blake, V. deMey, and X. Pintado, editors. *Object-Oriented Programming for Graphics*. Springer-Verlag, Berlin, 1995. To appear early 1995.

[10] M. Levoy. Spreadsheets for images. In *SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference, pages 139–146, Orlando, Florida, 24-29 July 1994. ACM SIGGRAPH.

[11] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[12] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[13] *SIGGRAPH '93*, Computer Graphics Proceedings, Annual Conference, Anaheim, California, 1-6 August 1993. ACM SIGGRAPH.

[14] P. S. Strauss and R. Carey. An object-oriented 3d graphics toolkit. *Computer Graphics*, 26(2):341–349, July 1992. SIGGRAPH'92, Chicago.

[15] J. J. van Wijk and R. van Liere. An environment for computational steering. Technical Report CS-R9448, Centrum voor Wiskunde en Informatica—CWI, Amsterdam, 1994.

[16] R. C. Zeleznik, K. P. Herndon, D. C. Robbins, N. Huang, T. Meyer, N. Parker, and J. F. Hughes. An interactive 3d toolkit for constructing 3d widgets. In SIGGRAPH '93 [13], pages 81–84.

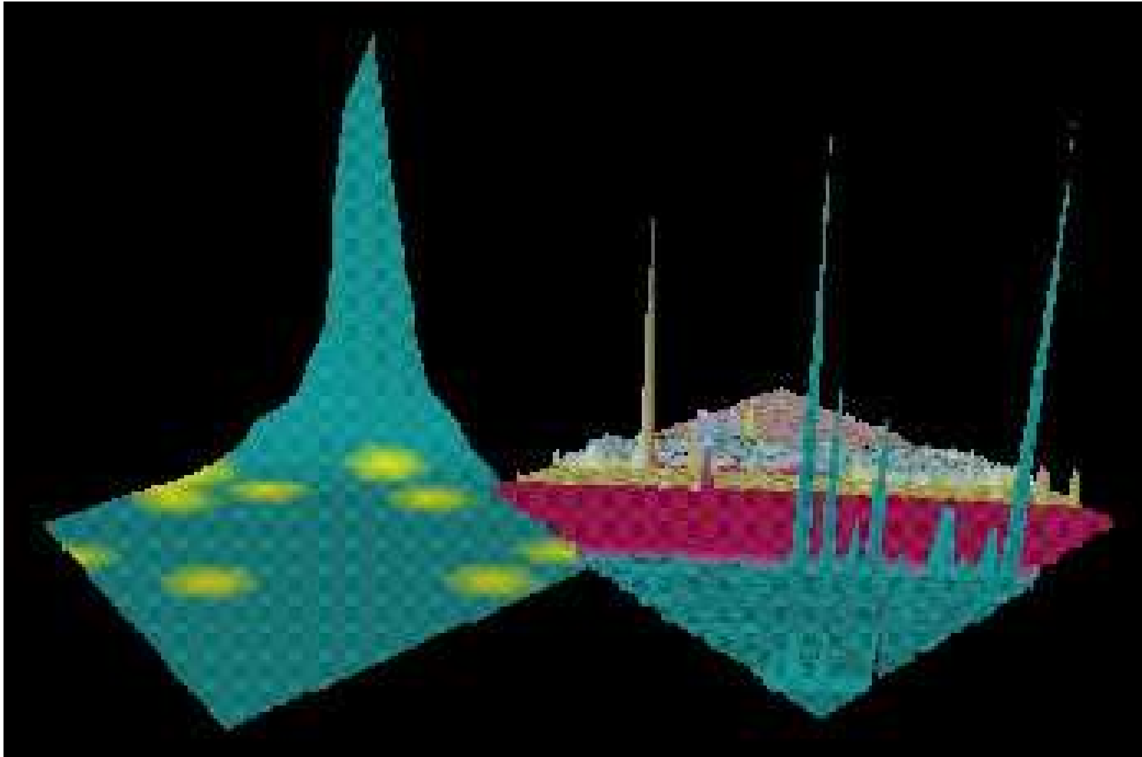

Fig. 4: Chiron source and object cost view.
The left surface represents the memory reference overhead incurred by each source line, while the right represents the memory reference overhead of each object used in the program. The left surface shows source lines that reference a particular object selected on the right surface.

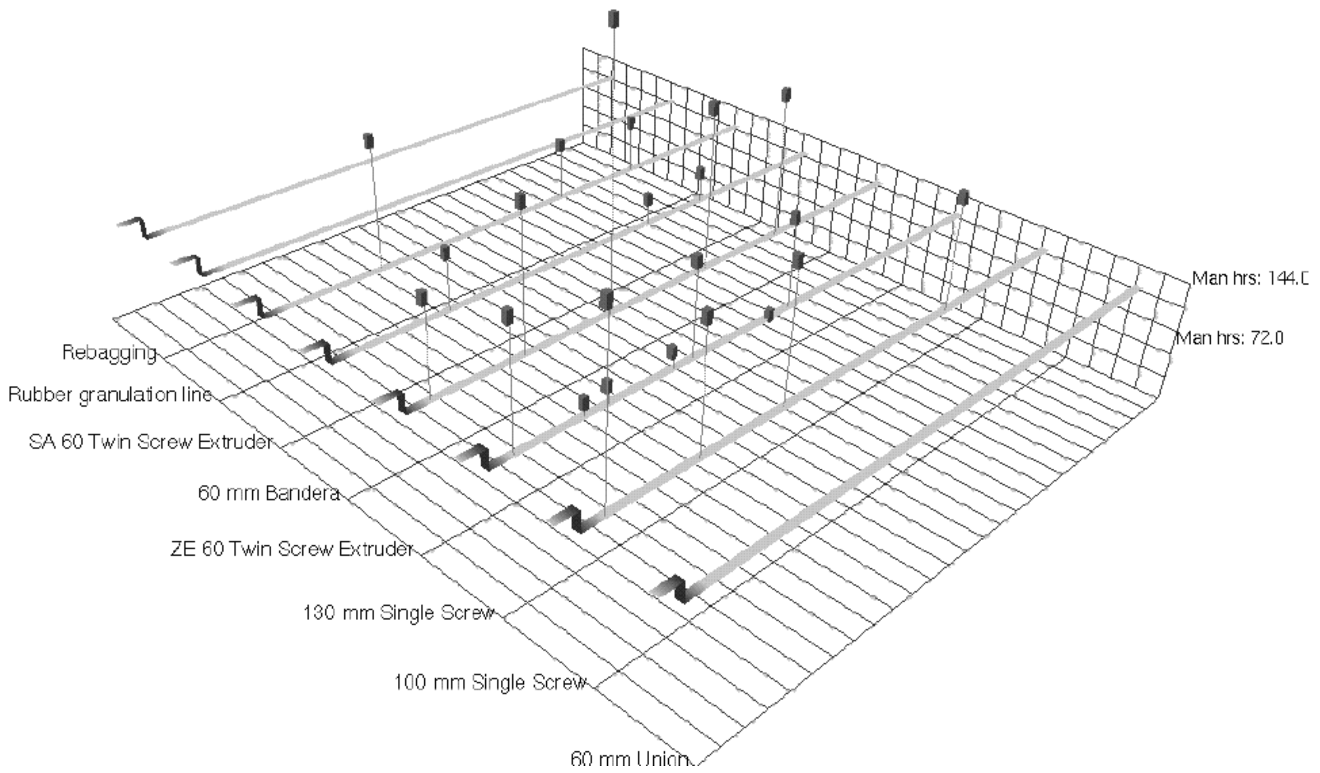

Fig. 5: Manufacturing visualization.
The rough-cut capacity planning module. The ribbons represent the capacity of each work center, while the cubes represent work centers where orders exceed capacity.