Viz: A Visualization Programming System

Harold H. Hersey, Steven T. Hackstadt, Lars T. Hansen, and Allen D. Malony
Department of Computer and Information Science
University of Oregon, Eugene, OR 97403
ph: 541-346-4407, fax: 541-346-5373
{hersey,hacks,lth,malony}@cs.uoregon.edu
April 1996

ABSTRACT

This paper describes the design and implementation of a high-level visualization programming system called Viz. Viz was created out of a need to support rapid visualization prototyping in an environment that could be extended by abstractions in the application problem domain. Viz provides this in a programming environment built on a high-level, interactive language (Scheme) that embeds a 3D graphics library (Open Inventor), and that utilizes a data reactive model of visualization operation to capture mechanisms that have been found to be important in visualization design (*e.g.*, constraints, controlled data flow, dynamic analysis, animation). The strength of Viz is in its ability to create non-trivial visualizations rapidly and to construct libraries of 3D graphics functionality easily. Although our original focus was on parallel program and performance data visualization, Viz applies beyond these areas. We show several examples that highlight Viz functionality and the visualization design process it supports.

1. Introduction

Visualization, as a *process* for creating meaningful visual representations of data, increases continually in sophistication with respect to techniques and methodologies as well as with respect to the diversity of the problem domains of its application. This has as much to do with what many believe to be the fundamental role of visualization in problem solving (the ability to improve data understanding) as it does with a greater insight into the underlying *mechanisms* of visualization operation (perception, cognition, data models, etc.). In recent years, however, the requirements placed on visualization technology--as a result of greater application needs and pressure to incorporate new visualization results--are outpacing the capabilities of existing tools. Indeed, the grand challenge problems facing visualization software stated over four years ago [22] remain problems today; if anything, they have increased in complexity. Not only are these challenges broad and interdisciplinary by nature, but proposed solutions are difficult to measure objectively [5,18]. Often the development of visualization technology must occur at the confluence of leading-edge research in several fields (*e.g.*, computer graphics, virtual reality, human-computer interaction, databases, expert systems) without a clear determinant that the *tool* produced can be effectively applied to meet users' application needs. What, then, are the primary factors that should drive visualization software design?

Without over-generalizing, there are three dominant concerns with present visualization technology:

1. the need to better integrate analysis and visualization through more "knowledgeable" data mapping;
2. the need to better apply user preferences, human perception, and problem requirements in visualization design, development, and use; and
3. the need to have a visualization system evolve/extend to include new techniques, data models, and application requirements.

Each of these concerns fundamentally involves issues of improving visualization meaning (*i.e.*, semantics) and, consequently, places the burden on visualization technology to somehow support semantics in its design. Because visualization problems vary, the priority and type of semantic support in tools must vary as well; a truly generic tool is not possible. This would suggest an approach to visualization software design that attempts to find effective, flexible mechanisms for including semantics into a visualization system. Similarly, each of the above mentioned concerns relates to aspects of the visualization process: data mapping, user/domain representation, and system operation. It is still premature to describe all the components of a visualization system and how they interoperate. Rather, visualization software based on "reference models" [2] or "abstract architectures" [22] may better allow process issues to be studied and supported.

We believe that the concerns stated above can best be addressed through a *programming system approach* to building visualization software. In this paper, we propose a visualization programming system called *Viz*. Viz was created out of a need to support rapid visualization prototyping in an environment that could be extended by creating programming abstractions tied to the application problem domain; our original application domain was parallel program and performance data visualization [7,8,9,10]. Viz provides this support in a programming environment built on an interpreted, high-level language with object extensions; it uses Scheme [12] augmented with a variation of the Common Lisp Object System [20]; Viz also provides an embedded 3D graphics library (Open Inventor [23]) for high-end graphics creation. In this environment, the developer can implement domain-specific analysis and visualization abstractions through procedural abstraction and object inheritance. A visualization process model called *data reactivity* is used to build mechanisms that have proved to be important in visualization design and operation (*e.g.*, constraints, controlled data flow, dynamic analysis, animation). The strength of Viz is in its ability to create nontrivial visualizations rapidly and to construct libraries of 3D graphics functionality easily. However, the ability to extend the Viz system with problem-specific programming abstractions, data models, and system interfaces allows it to be applied to areas beyond parallel program and performance visualization.

In this paper, we describe the Viz programming system. We begin in Section 2 with a discussion of the underlying data reactive model on which Viz is based. In Section 3, we present the Viz design and implementation. Section 4 provides several examples of Viz use that point out specific Viz features. Section 5 describes related work. Section 6 offers concluding remarks and thoughts on future directions.

2. Data Reactivity

2.1. Model

As a programming system, Viz must embody constructs and operational aspects that are fundamental to visualization creation and use. There are three important components of a visualization system that we believe must be supported in Viz: operations that describe mappings between data and graphics [2], constructs to specify and control dynamic behaviors (both synchronous and asynchronous) [21], and mechanisms to create high-level visualization objects that contain data mapping abstractions and dynamic behavior; these components are similar to those found in Butler's *abstract visualization machine architecture* [22]. Viz founds these components on an underlying reference model called *data reactivity*. Data reactivity serves as a model for how visualization objects are encapsulated (data and graphics) and interconnected (behavioral relationships). Data reactivity also supports an execution model based on functional evaluation. Data reactive objects holding evaluation functions are connected by data and control links to form a directed, acyclic execution graph with primitive input and output nodes at the fringes. With this foundational model, we believe the Viz system provides a robust programming environment for visualization development. Below, we describe the data reactive model in more detail.

Figure 1 is a diagram of a simple data reactive graph that defines a cube object for a bar graph visualization; the cube is translated in the *y*-direction by half its height, placing its base at *y*=0. There is one input node (an *accessor*) called `height`; this accessor maintains the height of the cube. Initially the height is 5 units. The output of the `height` node is connected to the inputs of two other nodes: to a primitive output node (a *field*) holding the height value for the cube object; and to the input of an interior node (an *engine*) that divides its input by 2. The output of the latter is connected to the `y` input of an engine that produces a transformation vector, the output of which is connected to a field in a transformation object. Finally, the transformation and the cube are grouped together in a scene graph.

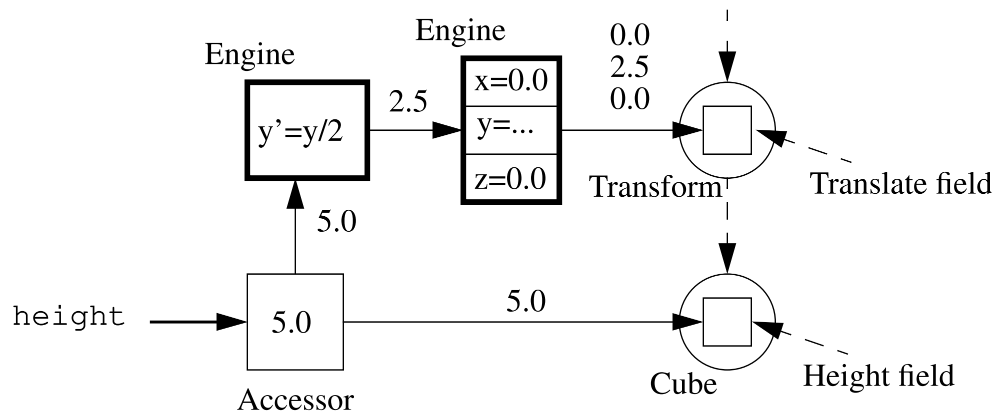

Figure 1. The data reactive network for a bar in a bar graph.

In the most typical mode of computation, the evaluation function in an engine is executed only on demand, when the value of a field which depends on the engine's output is demanded and any of the engine's input values have changed. If the value of the field is not demanded, no functions are executed regardless of whether input values have changed. The concrete upshot of this is that if a particular graphical object does not need to be rendered because it is outside the viewing frustum, then the object's fields will not be accessed by the renderer, and engines producing values for the fields will not be executed. Evaluation models will be covered in more detail below.

The graph in Figure 1 was created by the following Viz program:

```
(define height (accessor-node 5.0))
(define translation-engine
  (compose-vec3f :x 0.0
                 :y (eval-engine :input height
                                 :expr  (lambda (y) (/ y 2.0)))
                 :z 0.0))
(define my-scene
  (group (transformation :translation translation-engine)
         (cube           :height      height)))
```

(Program 1)

The procedure `compose-vec3f` creates an engine which produces a `vec3f`, a vector of length 3 suitable for use in a transformation operation, and `eval-engine` creates a simple one-input, one-output engine that computes the given function. The `group` procedure creates a scene graph; in the above program the arguments to `group` are a transformation node and a cube node. The data and control links are created implicitly.

2.2. Components

The components of the data reactive model are input nodes, output nodes, internal nodes, and connections.

Input nodes are called *accessors*. They are containers for values; any Scheme value may be stored in an accessor. The value stored in an accessor can be changed with the `set-value!` procedure and retrieved with the `get-value` procedure.

Output nodes are called *fields*; these are objects which will produce a value when asked to do so. Fields are used by Open Inventor to store attributes of graphical objects (see Section 3). A chain of engines can therefore terminate in a field "inside" a graphical object, and as explained below, this feature makes it possible to effect changes to a scene by storing new values in accessors.

The internal nodes are collectively called *engines*. An engine has one or more inputs, one output (which can be connected to several nodes), internal state, and an evaluation function which maps the inputs and the state to the output and a new state.

The data reactive model can have a number of different evaluation models; as a matter of implementation, Viz defines two: lazy and eager. The overall structures of the models are similar, so we discuss lazy evaluation first and then point out how eager evaluation differs.

In the lazy model, the values requested from fields force recomputation as necessary based on changes to accessors. When a new value is stored in an accessor, a control signal is sent to every node to which the accessor's output is connected. The control signal causes the receiver to be aware of the change in the accessor. The signal is then propagated to the dependents of the receiver, and so on, until all transitive dependents of the accessor node have been notified. No recomputation takes place at this time.[1] If a field has received notification of any changes to accessors on which its value depends at the time the value is requested, a new value will be requested from the engine or accessor from which the field receives its value. If that source node is an engine, the engine will recompute its value after first retrieving new input values from its source nodes, causing a cascading recomputation of all engines between the field and all accessors on which the field's value depends. The function in an engine is only recomputed once if its inputs have changed, even if its value is demanded from multiple dependents.

Consider the network shown in Figure 1 and Program 1. If we change the `height` accessor by storing a new value in it:

```
(set-value! height 7.0)
```

we get the situation in Figure 2(a); the value stored in the accessor has changed and each node depending on the value stored has received a change notification message (denoted by the asterisk), but the values stored in the engines and the fields in the network have not changed. When the scene is about to be rendered, both the `height` field in the cube object and the `translate` field

1. Note that a change to a mutable value stored in an accessor is not detected; the change must be to the accessor itself.

in the transformation object are flagged as out-of-date, and recomputation of the engine output values therefore has to take place. The resulting network is shown in Figure 2(b).

In the eager evaluation model, an engine's value is recomputed as a side effect of the engine receiving a change notification message from one of its inputs, before the change notification message is sent to the engine's dependents. If an engine has multiple inputs, it will receive change messages from its source nodes independently, and it may perform a recomputation before all its inputs have been recomputed, thus using some old values. This is in contrast to a topological evaluation order where all inputs are recomputed before the engine receives any change notifications.

It is possible to mix the lazy and eager evaluation models in a graph; this may not strike one as very useful but it makes it possible to connect larger data reactive components built from graphs which internally use either evaluation mechanism.

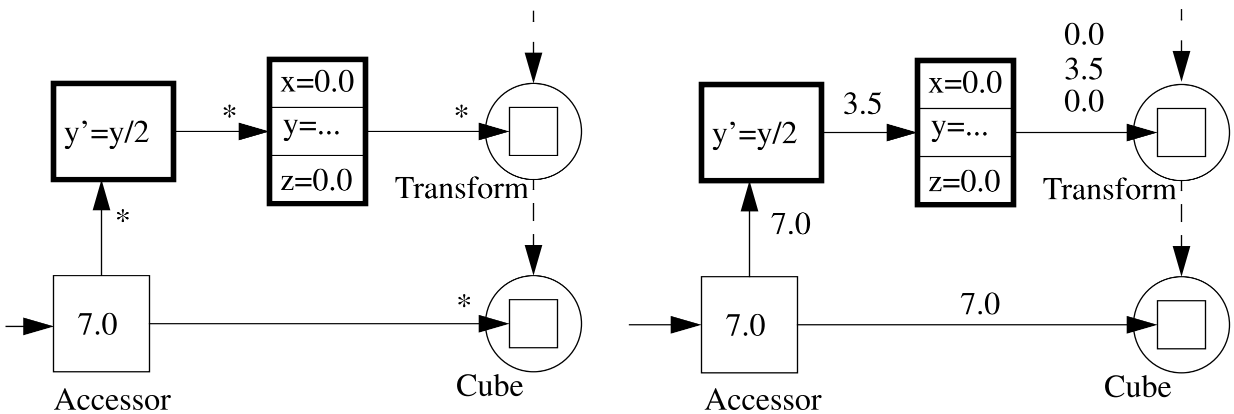

Figure 2. Lazy value propagation in the data reactive network: (a) after `set-value!` but before recomputation; (b) after recomputation.

2.3. Programming

The lazy evaluation model of data reactivity is similar to lazy evaluation as used in some functional programming languages [1,16] and has many of the same advantages: values are computed only when needed, long or unbounded data streams are easily accommodated, and filtering and pipelining are natural parts of the model. By using the eager evaluation model throughout, data reactive systems can emulate some data-flow systems.

A data reactive engine has state and can, for example, contain dynamic analysis functions that trigger the engine's output when thresholds are reached or unexpected or anomalous data are seen. In addition, asynchronous input sources are also accommodated in the data reactive model. An example is a network connections which delivers data at irregular intervals; by having the network deliver new values into an accessor node, the data reactive network can propagate the values to the consumer easily, transforming it along the way if necessary.

3. Viz

Viz is a programmable framework that supports both the visualization design process and the development of visualization tools. To address the concerns identified in Section 1, Viz provides an environment in which application semantics can be incorporated into the visualization design and creation process. The data reactivity model, described in the previous section, is key in accomplishing this.

Data reactivity manifests itself in Viz in two orthogonal ways: as a programming model, and as an implementation model for Viz itself. When programming in Viz, data reactivity acts as an abstraction of the data and computations that are used to create visualizations. With respect to the implementation of Viz, data reactivity essentially forms the interface between a high-level, interactive programming language and a 3D graphics library, the two major components of the Viz framework. The details of each aspect of data reactivity will be discussed separately. This section will focus on how Viz is implemented. Section 4 will demonstrate how data reactivity is used in building visualizations.

Figure 3 depicts the layered architecture of the components that make up Viz. The bottom three layers in the figure show the interlocking connections between the primary Viz components: Open Inventor and Scheme. The data reactivity layer connecting them represents the tight binding (*i.e.*, interface) between these components. To accomplish this binding, Scheme and Open Inventor have both been extended in certain ways. Collectively, these extensions effect the data reactive nature of Viz. The following subsections will first discuss the components and then explain how their extensions combine to create data reactivity.

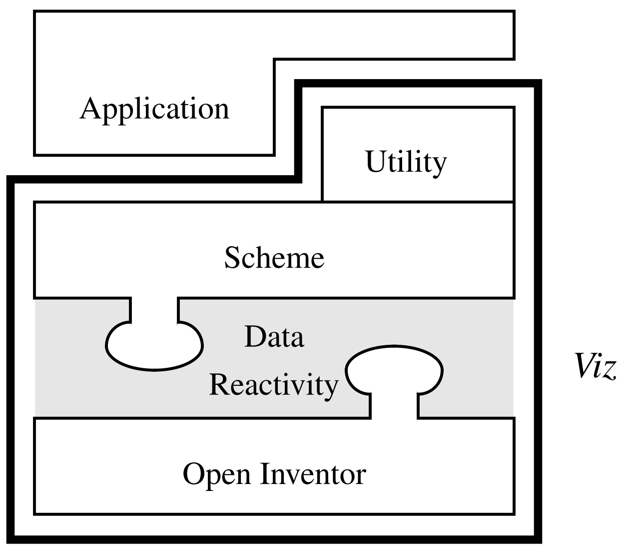

Figure 3. Viz implements data reactivity through extensions to language and graphics components. A utility layer supports the creation of user applications.

3.1. Open Inventor

Beneath the data reactivity layer is Open Inventor [23], a 3D graphics library that provides support for object oriented graphics creation and manipulation via a C++ class hierarchy. Open Inventor models a 3D scene in a structure called a *scene graph* [23]. A scene graph is a tree of *nodes* representing graphical objects, graphical properties, materials, transformations, cameras, lights, and so on. Of particular interest are the parameters of nodes, called *fields*. In the bar graph example of Section 2, a cube is realized as a graphical object node in an Open Inventor scene graph. One field of the cube node is its height. Similarly, a sphere node has a radius field. Open

Inventor uses *engines* to animate parts of a scene or to constrain one part of a scene to another; engines are also described in Section 2.

3.2. Scheme

On the upper side of the data reactivity layer is Scheme [12], a high-level, dynamically typed, interactive language. Scheme's memory management capabilities makes it particularly appropriate for managing the dynamic data structures--such as scene graphs--used in visualization. Scheme also provides a powerful abstraction mechanism with its lexical closures. Since Open Inventor has an object-oriented interface, we found it desirable--with respect to both code reuse and expressive power--to extend that interface into Scheme by using an object-oriented interface on the Scheme side. We have used the STk Scheme system and its object system STklos [6]; STklos is based on the Common Lisp Object System (CLOS) [20].

3.3. The Component Extensions - Data Reactivity

Implementing data reactivity required us to extend Scheme and Open Inventor in certain ways. The binding between Open Inventor and Scheme is achieved by *shadowing* Open Inventor classes and objects as STklos classes and objects on the Scheme side. Every Open Inventor object is shadowed by a corresponding STklos object, and changes made to an STklos object effects appropriate changes in the corresponding Open Inventor object. However, STklos objects maintain no state. Rather, an STklos object that mirrors an Open Inventor object invokes member functions in the latter to get or set the instance variables of the Open Inventor object. Low-level routines to access the Open Inventor library were implemented in C++ and are called from Scheme. Based on its ability to shadow Open Inventor objects in Scheme, Viz maintains a shadow of the Open Inventor scene graph.

On the Open Inventor side, a new type of node, called an *accessor node*, was derived from the Open Inventor class `SoNode`. An accessor node has a single field that can contain a Scheme value and that can be connected to any other Open Inventor field. Accessor nodes, like engines, reside outside the scene graph. Since accessor nodes contain Scheme values which may not be directly compatible with Open Inventor, Viz inserts *type converters*, special engines that convert between Scheme and Open Inventor data types, along connections from accessor nodes to other nodes. For example, a Scheme vector of length 3 can be converted to an Open Inventor `vec3f`. Accessor nodes have a shadow representation on the Scheme side.

Another key extension to Open Inventor is a new type of engine called an *eval-engine*. Eval-engines are Open Inventor engines whose inputs are Scheme values and whose evaluation functions are Scheme procedures. The procedure in an eval-engine can be defined and redefined at run-time, giving the Viz programmer considerable flexibility. Eval-engines are also shadowed by STklos objects.

These ideas--shadowing, accessor nodes, eval-engines--come together in two structures: a data reactive network of accessor nodes and engines, and a scene graph of the Open Inventor objects described earlier. The two structures are connected wherever the output of an accessor node or an engine connects to a field of an Open Inventor object, as illustrated in Figure 1.

3.4. Utility Layer

The utility layer, written in Scheme, provides a set of basic objects and procedures that simplify common tasks such as the composition and animation of graphical objects. For example, users can specify the starting and ending positions for an object, and animation support in the utility layer will automatically compute the appropriate interpolation for a smooth transition between the two (see Section 4.3). The utility layer is intended to be the starting point for building user visualizations and application-specific visualization environments.

4. Examples

In this section we give several examples that demonstrate certain features of Viz and how one might program using Viz.

4.1. Bar Graph

A *bar graph* is a collection of individual bars aligned along an axis, where the height of each bar is proportional to some data value. In Section 2, we introduced the beginnings of a bar graph visualization by showing how an individual bar is constructed and its height value changed to effect a change in the graphics. A bar graph can be created in Viz as the composition of individual bars using the `make-bar-graph` function defined below. (The `make-bar` function referenced in Program 2 is similar to the code in Program 1.)

```
(define (make-bar-graph data-vect spacing)
  (let* ((heights   (vector-map accessor-node data-vect))
         (bar-graph (vector-map (lambda (node)
                                  (group
                                    (transformation  :translation spacing)
                                    (make-bar node)))
                                heights)))
    (values
      (separator bar-graph)
      heights)))
```

(Program 2)

This function first wraps the incoming data elements (`data-vect`) in separate accessor nodes, all of which are stored in the `heights` vector. It then makes and aligns a single graphical bar for each accessor in `heights`, creating a vector of scene graphs (`bar-graph`). Finally, a single scene graph is created by the call to `separator`, and that scene graph and the vector of accessor nodes are both returned. The accessor nodes for the heights of the bars can now be changed, and the graphical representation of the bar graph will change accordingly. The top figure in Image 1 shows a bar graph, while the bottom figure shows another level of composition, where bar graphs are composed into a 2D grid of bars.

4.2. Surface Plot

Viz can be used to extend the set of graphical objects provided by the Open Inventor 3D graphics library without having to extend Open Inventor itself by coding new C++ classes. In contrast to how the bar graph visualization is created, we developed a new "primitive" graphic object, a *surface plot*, in Viz to present mesh data. Open Inventor does not have a primitive surface plot, but instead has a generic graphic facility for creating triangle strip sets (*i.e.*, collections of polygons). The Viz surface plot object creates a characteristic indexed triangle strip set for displaying surfaces that can be accessed through a single accessor node containing all of the mesh values. This accessor node can be used to control the surface plot fields in a manner similar to basic Open Inventor graphic objects. Image 2 shows three surface plot visualizations that come from a High

Performance Fortran program simulating a vibrating membrane. The main point here is to realize that Viz can accommodate different granularities of accessors and can be used to create new graphic objects.

4.3. Rubix Cube

A visualization of the Rubix Cube was conceived at an early stage in Viz's development as a motivating example for providing control over parts of a visualization that depends on changing state in a dynamic simulation. Representing the Rubix Cube data state and creating the graphics for that state are trivial problems. The data consists of a vector of 27 pairs, each representing a cube and the direction it is facing. The graphics are simply 27 small cubes, each with 6 differently colored faces, arranged in a 3x3x3 cube. The difficulty in visualizing a Rubix Cube is in the dynamic mapping for repositioning and animating the individual cubes. When a move is performed, the cubes are repositioned to their next state through an animation. This occurs in two phases. In the first phase, eval-engines are used to determine the next position and orientation of each cube, as reflected in the accessor structures. The second phase is to create the animation as a coordinated movement of the nine cubes affected by the move. This is accomplished with an engine that creates a sequence of intermediate cube positions by changing the angle of rotation about a common center point and axis for the selected cube set. Image 3 shows the cube in an intermediate position as the left face is rotated. Implementing this behavior by programming Open Inventor directly would be significantly more difficult.

4.4. VizMan

Our last example is used to show how Viz can be used to build up visualization abstractions. We chose to create a visualization of a running man, VizMan, built from articulated joints. A joint function was developed which generated an engine for each human joint based on parameters for its field of motion. The joints were then connected together in a `body` abstraction (see the code segment in Program 3) by creating a motion hierarchy in a scene graph. The body could then be set in motion by connecting all of the engine inputs to a single clock value. As the clock increases, the head, arms, legs, torso, ankles, and body move. VizMan in stride is shown in Image 4.

```
(define (body)
  (separator
   ; vertical oscillation
   (transform :translation
              (compose-vec3f :x 0 :y (vertical-oscillation) :z 0))
   ; lean forward a bit
   (transform :rotation (compose-rotation :axis #(1 0 0) :angle 0.2))
   ; upper torso
   head neck (shoulder-girdle) chest
   ; right and left arm
   (separator (rshoulder) ruparm (relbow) rlowarm (rwrist) rhand)
   (separator (lshoulder) luparm (lelbow) llowarm (lwrist) lhand)
   ; lower torso
   (pelvic-girdle) pelvis
   ; right and left leg
   (separator (rhip) rthigh (rknee) rshin (rankle) rfoot)
   (separator (lhip) lthigh (lknee) lshin (lankle) lfoot)))
```

(Program 3)

5. Related Work

Viz can be regarded as a confluence of ideas that have arisen in the context of research work in high-level, interactive languages, object-oriented graphics, graphical animation systems, and visualization programming and application environments. Below, we relate Viz to work from these areas.

Perhaps the most directly related works from the standpoint of graphics programming are SGI's Open Inventor [23] and Russell's Ivy [19]. Viz clearly benefits from Open Inventor's object-oriented graphics model and functionality: primarily, scene graphs, basic graphic objects and primitives, and engines. Viz serves to extend these features into a Scheme environment to facilitate graphics programming. But Viz also attempts to address certain operational shortcomings of Open Inventor. A good example is animation. Engines are used in Open Inventor to implement animation by changing the state of property and shape nodes, but programming engines and chaining them together can be complicated. Viz makes engine creation easy by allowing engine functionality to be specified in Scheme (via eval-engines), by connecting engine outputs to fields in Scheme (via data reactivity), and then by generating the Open Inventor engines that support the specified functionality. As a result, quite complex engine-based animations, such as VizMan, can be more easily programmed than in Open Inventor. The other clear contrast of Viz to Open Inventor is Viz's interactive environment. It has been remarked that visualization development requires many design iterations and that interactive systems can better support rapid prototyping [14]. This is certainly true in the case of Viz.

Ivy is a Scheme interface to Open Inventor developed mainly to simplify Open Inventor programming. In this respect, Ivy provides some of the same benefits as Viz: an interactive environment for incremental construction of Open Inventor programs; the use of Scheme procedures in engines; and mechanisms for using Scheme procedures as Open Inventor callbacks. However, many of these mechanisms are at fairly low semantic levels. In contrast to Viz, Ivy provides little in the way of "value added" components. It does not provide engines that perform type conversion, nor does it have the facility for accessor nodes. Finally, there is no notion of a visualization model, such as data reactivity, built into Ivy.

Several research efforts have provided interactive facilities for 3D graphics programming and visualization, particularly for creating animations. One of these is Alice [4] which is a 3D graphical library embedded into the Python language. Another is Obliq-3D [14]. Alice has many functional similarities to Obliq-3D, which we will discuss below. The Actor/Scriptor Animation System (ASAS) [17] was one of the earliest to use an interactive language approach, in this case, based on the Actor paradigm and Lisp. Other examples of systems with interactive languages are SuperGlue [11] and IRIS Explorer [15].

Much of the philosophy, goals, and functionality embodied in Viz are also found in the work by Najork and Brown on Obliq-3D [14]. Obliq-3D is a language system for rapidly prototyping 3D animations, mainly algorithm animations. It combines a 3D graphics animation library, Anim3D [13], with the object-oriented, interpreted language Obliq [3], and with modules to construct 3D animations. Obliq is a powerful language that allows Obliq-3D to use one-way constraints as a mechanism for connecting graphical parameters, and to provide for hierarchical grouping of graphical objects. Obliq-3D supports asynchronous and synchronous behaviors; Viz captures these as engines. Viz provides perhaps more control over propagation than Obliq-3D, but Obliq-3D's animation support is more robust. Obliq-3D will also more naturally support distributed visualization, as Obliq is a distributed language. Viz will have to implement interfaces to distributed systems to provide this capability.

6. Conclusion and Future Work

Viz is a visualization programming environment that supports rapid visualization prototyping. It utilizes a Scheme language system with object extensions to realize interactive execution and provides sophisticated 3D graphics through an API to Open Inventor. Fundamentally, Viz is based on a data reactive model of visualization operation; data reactivity captures important concepts in support of data mapping, reactive data and control behaviors, dynamic graphical updating, and interaction. The data reactive model is implemented as low-level extensions of Open Inventor functionality and high-level programming abstractions that facilitate a programmer's ability to rapidly develop visualization prototypes. Viz has been shown to significantly reduce the amount of code needed to create Open Inventor graphics (often by a factor of two or greater). Viz can be easily extended with APIs to other libraries or toolkits; for instance, we intend to build a Viz module for inclusion in IRIS Explorer.

Presently, Viz is a research system that has been used to construct demonstrations that test its capabilities. Clearly, our main concern in the near term is to deploy Viz in application scenarios where specific visualizations are needed. For instance, we have been collaborating with geologists on a seismic tomography project that uses Viz to create visualizations like those in Image 5. These displays show seafloor earth models and the tomographic imaging ray paths used to refine these models. We are also actively defining new abstraction layers to support, for instance, interaction and animation abstractions. In the longer term, we will be extending Viz to operate in distributed environments.

References

[1] H. Abelson and G. J. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[2] D. Butler, J. Almond, R. Bergeron, K. Brodlie, and R. Haber, *Visualization Reference Models*, In Proc. Visualization '93, Panel Discussion, IEEE Computer Society Press, Nov. 1993, pp. 337-342.

[3] L. Cardelli, *A Language with Distributed Scope*, In Proc. 22nd Annual ACM Symp. Principles of Programming Languages, Jan. 1995, pp. 286-297.

[4] M. Conway, R. Pausch, R. Gossweiler, and T. Burnette, *Alice: A Rapid Prototyping System for Building Virtual Environments*, Proc. ACM CHI '94 Conf. Human Factors in Computing, Conf. Companion, Apr. 1994, pp. 295-296.

[5] F. D. Fracchia, *Is Visualization Struggling under the Myth of Objectivity?*, In Proc. Visualization '95, IEEE Computer Society Press, Panel Discussion, Nov. 1995, pp. 412-415.

[6] E. Gallesio, *STklos: A Scheme Object Oriented System Dealing with the Tk toolkit*, In Proc. of the USENIX 1994 Symposium on Very High Level Languages.

[7] S. Hackstadt and A. Malony, *Next-Generation Parallel Performance Visualization: A Prototyping Environment for Visualization Development*, In Proc. Parallel Architectures and Languages Europe (PARLE) Conference, Athens, Greece, Jul. 1994, pp. 192-201.

[8] S. Hackstadt, A. Malony, and B. Mohr, *Scalable Performance Visualization for Data-Parallel Programs*, In Proc. Scalable High Performance Computing Conference (SHPCC), Knoxville, TN, May 1994, pp. 342-349.

[9] M. Heath, A. Malony, and D. Rover, *Parallel Performance Visualization: From Practice To Theory*, IEEE Parallel and Distributed Technology, Vol. 3, No. 4, Winter 1995, pp. 44-

[10] M. Heath, A. Malony, and D. Rover, *The Visual Display of Parallel Performance Data*, IEEE Computer, Vol. 28, No. 11, Nov. 1995, pp. 21-28.

[11] J. Hultquist and E. Raible, *SuperGlue: A Programming Environment for Scientific Visualization*, In Proc. Visualization `92, IEEE Computer Society Press, Nov. 1992, pp. 243-250.

[12] *IEEE Standard 1178-1990. Standard for the Scheme Programming Language*, IEEE, New York, 1991.

[13] M. Najork and M. Brown, *A Library for Visualizing Combinatorial Structures*, In Proc. IEEE Visualization ‘94, Oct. 1994, pp. 164-171.

[14] M. Najork and M. Brown, *Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System,* IEEE Trans. on Visualization and Computer Graphics, Vol. 1, No. 2, June 1995.

[15] Numerical Algorithms Group, Ltd., *IRIS Explorer User’s Guide, Release 3.0*, NAG Ltd., 1995.

[16] S. L. Peyton-Jones, *The Implementation of Functional Programming Languages*, Prentice/Hall International, Englewood Cliffs, NJ, 1987.

[17] C. Reynolds, *Computer Animation with Scripts and Actors*, Computer Graphics, Vol. 16, No. 3, July 1982, pp. 289-296.

[18] H. Rushmeier, *Metrics and Benchmarks for Visualization*, In Proc. Visualization ’95, IEEE Computer Society Press, Panel Discussion, Nov. 1995, pp. 422-425.

[19] K. Russel, *Ivy, A Scheme Binding for Open Inventor*, available at http://www-white.media.mit.edu, Mar. 1996.

[20] G. L. Steele, Jr., *Common Lisp: the Language. Second edition,* Digital Press, Bedford, MA, 1990.

[21] N. Thalmann and D. Thalmann, *Computer Animation: A Key Issue for Time Visualization*, Scientific Visualization, Academic Press, 1994, pp. 201-222.

[22] L. Treinish, D. Butler, H. Senay, G. Grinstein, and S. Bryson, *Grand Challenge Problems in Visualization Software*, In Proc. Visualization ’92, IEEE Computer Society Press, Nov. 1992, pp. 366-371.

[23] J. Wernecke, *The Inventor Mentor*, Addison-Wesley, Reading, MA, 1994.