An Environment for Computational Steering

Jarke J. van Wijk [1,2]
Robert van Liere [2]

August 6, 1996

Abstract

Computational Steering is the ultimate goal of interactive simulation: researchers change parameters of their simulation and immediately receive feedback on the effect. We present a general and flexible environment for computational steering. Within this environment a researcher can easily develop user interfaces and 2-D visualizations of his simulation. Direct manipulation is supported, and the required changes of the simulation are minimal. The architecture of the environment is based on a Data Manager that takes care of centralized data storage and event notification, and satellites that produce and visualize data. One of these satellites is a graphics tool to define a user interface interactively and to visualize the data. The central concept here is the Parametrized Graphics Object: an interface is built up from graphics objects whose properties are functions of data in the Data Manager. The scope of these tools is not limited to computational steering, but extends to many other application domains.

Keywords: Visualization, computational steering, interaction, direct manipulation.

1 Introduction

1.1 Computational Steering

Scientific Visualization has become a major research area since 1987, when the influential report of the NSF [1] was published. In recent years many new methods, techniques, and packages have been developed. Most of these developments are limited to post-processing of data-sets. Usually the assumption is made that all data is generated first and that next the researcher iterates through the remaining steps of the visualization pipeline (selection, filtering, mapping, and rendering) to achieve insight in the generated data. Hence, the interaction with the simulation is limited.

Marshall et al. [2] distinguish two alternatives to this post-processing approach to visualization. The first step to more interaction with the simulation is with *tracking*. After each time-step of the simulation the resulting data for that time-step is sent into the visualization pipeline and can be inspected. If the researcher considers the results invalid, then the simulation can be stopped at an early stage, and restarted with a different set of input parameters. The next step, *Computational*

[1] Netherlands Energy Research Foundation ECN, P.O. Box 1, 1755 ZG Petten, The Netherlands.
[2] Centrum voor Wiskunde en Informatica, P.O. Box 4097, 1009 AB Amsterdam, The Netherlands.

Steering goes a lot further, and can be considered as the ultimate goal of interactive computing. Computational steering enables the researcher to change parameters of the simulation while the simulation is progressing. According to Marshall et al. [2] : "Interaction with the computational model and the resulting graphics display is fundamental in scientific visualization. Steering enhances productivity by greatly reducing the time between changes to model parameters and the viewing of the results."

Our aim is to provide researchers with a Computational Steering Environment (CSE) that encourages exploratory investigation by the researcher of his simulation. In the following subsections we sum up the requirements, we discuss existing solutions, and give an overview of the remainder of this paper, in which we present our solution.

1.2 Requirements

The data-flow between the researcher and his simulation via a CSE is shown in figure 1. The researcher can enter new values for parameters, and views visualizations of the resulting data. Hence, input widgets such as text-fields, sliders, and buttons must be provided, as well as a variety of visualization methods, such as graphs, text, graphics objects, etc. With such objects input and output are separated. For more direct control, objects must be provided that allow for two-way communication: both input and output. It must be possible to select and drag visualization objects, thereby directly controlling parameters of the simulation. In other words, direct manipulation must be provided.

The simulation receives from the CSE new parameter values, and sends newly calculated results to the CSE . We assume that the simulation can handle changes of parameters on the fly, and that it can provide meaningful intermediate results within a time-interval that is acceptable to the researcher. The concept of direct manipulation has a counterpart in the interface between CSE and simulation. Some variables, typically state-variables, are continuously updated by the simulation, but can also be changed from outside of the simulation.

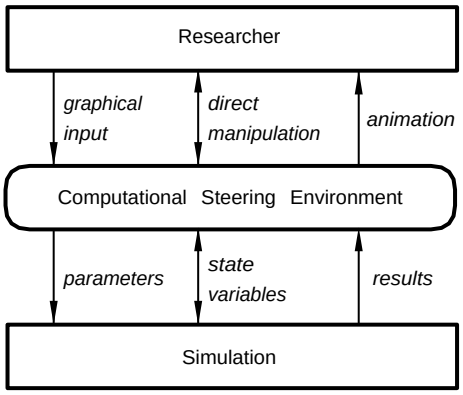

Figure 1: Data flow between researcher, CSE , and simulation

As an example of an interface with a high degree of interaction, consider figure 2. This shows an interactive graphics interface to a simulation of a set of bouncing balls in two dimensions. The balls are depicted as circles, each with a small red line that indicates the direction and magnitude

of its velocity. For every time step the simulation calculates the position and velocity of all balls. Various control-parameters such as the size of the balls, damping, attraction, a constant field-force, can be set. This can be done via sliders, by typing, or by dragging the arrows. The positions of the balls (state-variables) are continuously updated by the simulation, but can also be changed by the researcher by dragging their graphical representations to other positions. This figure is a screen-dump of a result of the CSE presented in this paper. In section 4.1 we come back to this example and show how it was defined.

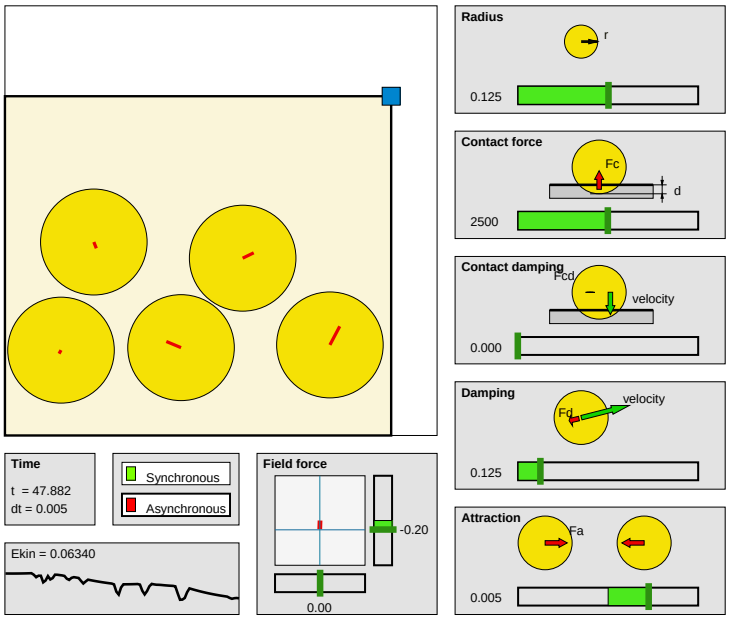

Figure 2: Simulation of bouncing balls

The researcher must be enabled to create and to refine the interface to the simulation easily and incrementally. The process of achieving insight via simulation is an incremental process. For all stages of the visualization pipeline (from simulation to rendering) the cycle *specification*, *implementation*, *application* is continuously reiterated. In the process of gaining insight via computational steering, a researcher typically wants to look at and to control other, possibly new variables, and to visualize them in different ways.

The CSE must be able to deal with multiple processes simultaneously. There are three reasons for this: First, it must be possible to integrate existing tools, such as special purpose packages for grid-editing or visualization, into the CSE . Second, a simulation is often built up of several processes running on distant compute servers. Third, it would be convenient if several researchers could view and control the data simultaneously.

The final requirement concerns the underlying data model and the amount of data within the CSE . The type of data to be handled depends very much on the type of simulation, and therefore can vary from simple scalar data to large, three-dimensional, time-dependent vector and tensor field data-sets. The underlying data model must be flexible enough to support a wide range of data types and data quantities.

1.3 Related work

An optimal result for the researcher can be achieved if we build a complete system, incorporating handling of input, output, as well as the simulation, using basic graphics libraries, such as for instance IRIS GL, or PHIGS. Typical examples are flight-simulators and packages for mechanical engineering. This approach requires a large effort and considerable experience, and often leads to an inflexible system with fixed functionality. Thus, the requirement for easy modification by the researcher is violated. However, if the application is time-critical and has wide-spread use, the gain can be worth the cost.

Given the set of requirements, how can we realize a more general solution, i.e. an environment rather than a turn-key system ? Basically, three approaches can be taken:

- consider the design and implementation of a CSE as a graphics problem, and extend a basic graphics library;
- consider it as a user interface problem and extend the functionality of a User Interface Management System (UIMS);
- consider it primarily as a visualization problem, and extend an application builder.

All these approaches have been pursued. An exhaustive treatment would require far more space then is available here, because very many related approaches, techniques and concepts exist. We limit ourselves therefore to some relevant examples.

Graphics libraries typically offer only low level functionality. To simplify the definition of interactive applications with direct manipulation, graphics libraries can be enhanced with higher level interactive objects. With this approach a tight coupling of application objects and interaction objects can be ensured. For example, the Inventor toolkit [3] provides an object-oriented library which simplifies the development of interactive graphics applications. Van Dam et al. [4] take this a step further. They describe an extensible system that primarily aims at the integration of a variety of simulation and animation concepts in the graphics toolkit. Objects may have geometric, algorithmic, or interactive properties. Objects may send messages to each other which, after being stored in the object, can be edited. The authors show how collision detection [5], constraints and deformation [6] are handled within this framework.

Various UIMS's have been described that provide support for coupling the application (represented by a set of application objects) and the user interface [7, 8, 9]. User interface and application designers can develop their parts independently and have the UIMS manage the dialogue layer to integrate the two parts. Some UIMS's allow users to specify direct manipulation interfaces through WYSIWYG editors. An example is the Peridot system [10], which allows non-programmers to create sophisticated interaction techniques. With the help of graphical constraints, Peridot allows a user to draw graphical objects of the user interface. The user provides the behavior of these graphical objects with a technique called *programming by example*. This results in Peridot generating parameterized procedures which can, in turn, be linked in or interpreted by the application program.

Application builders have emerged as a flexible solution for scientific visualization. The users are provided with a set of modules, which can be connected and extended to rapidly prototype applications and reconfigure existing ones. Some researchers have discussed the use of computational steering [11, 12, 2] in this context. With the current generations of such systems the user can define user interface panels. The simulation has to be included as a module. The

extension of such systems with direct manipulation has been discussed in [13]. In general, the implementation of direct manipulation is cumbersome, because in data flow environments the relation between the original data and the geometric objects in the visualization is not known.

Finally, a novel approach to exploratory visualization has been taken in VIEW [14]. The key idea of VIEW is a very tight coupling of geometry with an underlying database. The VIEW system allows researchers to interact directly with the visualization. Researchers can select tools for the visualization of their data. This allows and encourages researchers to experiment and explore the underlying data spaces. Scripting languages are used for defining new tools.

All approaches discussed have a value on their own right. However, we feel that none satisfies all CSE requirements simultaneously. The extensions discussed of basic graphics libraries are very convenient for graphics application programmers, but are not directly suitable for use by researchers. The visualization of data is outside the scope of current UIMSs. Application building environments do not provide direct manipulation, which we consider as an important issue for computational steering. In the VIEW system new visualization tools must be specified via a specialized scripting language. Here the graphics objects, and the relations between graphics objects, data, and user actions are specified.

1.4 Overview

In the following sections we present an environment for Computational Steering that does satisfy our requirements. The space available here does not allow for a detailed treatment, a more extensive description can be found in [15]. In section 2 we give an overview of the architecture. The central component is a Data Manager, which is responsible for managing data storage and process communication. In section 3 a general tool for graphical input and output is described. We have taken the approach of extending low level graphics. We use the notion of a Parametrized Graphics Object (PGO) as the main concept, and a graphics editor as a metaphor for the design of the interface. Examples of applications are given in section 4, followed by a discussion (section 5) and conclusions (section 6).

2 Architecture

Flexibility implies that the functionality is spread over several separate processes, for simulation, for input, output, as well as auxiliary operations on data. These processes must have access to the same set of data. We therefore use the architecture shown in figure 3. The central process is the *Data Manager* , the other processes we call *satellites* . Satellites can connect to and communicate with the Data Manager.
The purpose of the Data Manager is twofold:

- Manage a database of variables. Satellites can create and do read / write operations on variables. For each variable its name, type (*floating point, integer, string*) and its current value is stored and managed. Variables can be scalar variables or arrays, in which case also the number of dimensions and size of the dimensions is stored. These sizes can change dynamically.

- Act as an event notification manager. Satellites can subscribe to a set of predefined events. For example, if a satellite subscribes to mutation events on a particular variable, the Data Manager will send a notification to that satellite whenever the variable is updated.

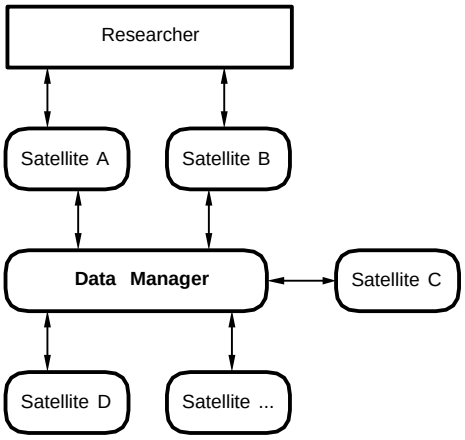

Figure 3: Data Manager and Satellites

The functionality of the Data Manager is purposely limited: it can be used by satellites for communication, but it does not control these satellites themselves. This is in contrast to data flow oriented application builders, in which the underlying execution models dictate when modules will be fired [16, 11, 17]. A number of small but useful general purpose satellites have been developed. With these satellites variables in the Data Manager can be updated. Some examples are :

- *dmdump*, *dmrestore* dump and restore the values of the variables to and from file;
- *dmslice* selects subsets of arrays;
- *dmlog* maintains a log of the last N values of a variable;
- *dmcalc* evaluates arithmetic expressions on variables;
- *dmscheme* is a Scheme interpreter which is extended with the API to the Data Manager;
- *ReadDM* and *WriteDM* are two IRIS Explorer modules which translate Data Manager variables into the Explorer data types *cxParameter* or *cxLattice* and back.

Communication of a satellite with the Data Manager is done via a small Application Programmers Interface (API). The abstractions used by this low level API are similar to standard UNIX I/O file handling, with variables instead of files. Satellites use handles to read, sample and write to variables. A sample call returns immediately, a read call waits until the value of a variable changes. Events are used to indicate state changes in the Data Manager. Various routines are provided to query the status of variables and the Data Manager.

The functionality provided by this low level API is compact, terse and complete, but not simple to use. Therefore, on top of this API a Data I/O library was defined, which is tuned to the needs of researchers that want to integrate their simulations within the CSE . The two main calls are `dioConnect` to register a variable whose value must be read and/or written, and `dioUpdate`, which call takes care of updating the variables in the simulation. Entry points in existing code where these calls must be added are generally easy to locate: connection of variables just before the main loop, and update of variables at the end of the main loop, just after the new results of

the time-step have been calculated. The application programmer does not have to change the control-flow of his simulation, and can use a procedural programming language.

3 Parametrized Graphics Objects

3.1 Overview

In this section a tool for the graphical interaction of a researcher with a simulation is described. As stated before, an important requirement for such a tool is that the visualization of the data, the handling of user input, as well as direct manipulation are provided. One way to solve this is to consider these aspects as disjoint, and to provide ingenious, but unrelated solutions. We have chosen a different solution: look for the greatest common divisor of these aspects, and provide an homogeneous solution. The greatest common divisor of user input widgets and visualization tools is simply graphics. Buttons, sliders, graphs, histograms all boil down to collections of graphics objects. Therefore, we use Parametrized Graphics Objects (PGOs) as the main concept, and the graphics editor as a metaphor for the design of the interface.

The graphics editor has two modes: specification and application, or shorter, *edit* and *run*. In edit-mode, the researcher can create and edit graphics objects much like in MacDraw-like drawing editors. The properties of those objects can be parametrized to values of variables in the Data Manager. Hence, the researcher draws a specification of the interface. In run-mode, a two way communication is established between the researcher and the simulation by binding these properties to variables. Data is retrieved and mapped onto the properties of the graphics objects. The researcher can enter text, drag and pick objects, which is translated into changes of the values of variables. What you draw is what you control. The working method of a researcher is thus:

1. Decide which parameters are important for control and visualization;
2. Adapt the simulation to connect those parameters with the Data Manager;
3. Edit an interface;
4. Run the interface: view and control the simulation;
5. Analyze the results and go back to one of the previous steps.

In addition, standard satellites can be used or new satellites can be developed. The graphics editor itself is just a satellite, as shown in figure 4. When the researcher interacts with PGOs , data will be written to the Data Manager. Similarly, writes by other satellites to variables will trigger the graphics editor and result in visualizations of the data.

Figure 5 shows that the researcher is free to design representations that convey the semantics of the underlying variables: Nine different ways to visualize two scalar variables x and y are shown Both the edit- and the run-mode versions of the interface are given. The variables can be presented via standard UI-widgets (a and b), or in a business graphics style (c). The parameters can also refer to a position (d), a range (e), or have a mechanical (f) or sensual (g) interpretation. Typical computer graphics interpretations are given in (h) and (i). Many more representations can be conceived.

Figure 6 provides an overview of the main objects within the PGO editor. For each object one or more examples of their presentation to the researcher are shown. Various Parametrized

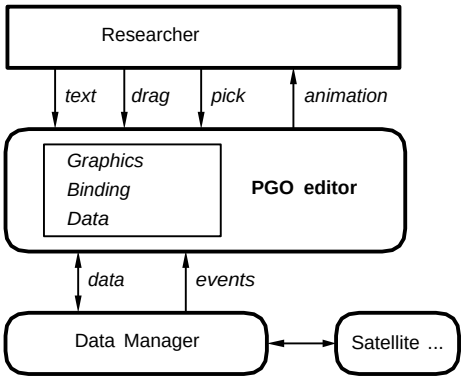

Figure 4: The PGO editor in the CSE

Graphics Objects (PGOs) are shown in the top row of the figure. The geometry of each of these objects is defined by points, the non-geometric properties are defined by various attributes. In the bottom row of the figure the objects for local data management are shown. The object responsible for the binding between graphics and data is the Degree Of Freedom, shown in the middle row.

In the following subsections we first describe each object in more detail. After that, we will describe how they interact when the editor is switched to run-mode. Finally, the handling of arrays is discussed. Applications are given in the next section.

3.2 Graphics objects

The PGO editor offers a set of standard graphics objects: fill-area, polyline, rectangle, circle, arc, and text. The geometry of the objects is defined by one or more points. These points are independent of the graphics objects themselves, so that one point can be shared by various graphics objects.

The researcher can define relationships between points (figure 7). Any point can have another point as a reference-point or parent-point. These relations are shown as grey lines with yellow arrows. Cycles are not allowed, thus, the structure is a forest of trees, with points as nodes and leaves. These relations are used when points are moved. How this is done depends on the type of the point. Two types of points can be used:

- Hinge points (depicted as circles, and labeled H). When a hinge point is moved, the same translation is applied to its child points.
- Fixed points (depicted as diamonds, and labeled F). If a fixed point is moved, then its children are rotated such that the angles between the points and their distances to the parent point remain fixed.

Next these transformations are recursively applied to the children of the transformed points. In figure 7 we show the effect of moving a point in run-mode along a Degree of Freedom.

Graphics objects have four attributes: the hue, saturation, and value of its color, and the linewidth used for the object or its outline. In text objects references to Mapped Variables can be made by using a $, followed by a Mapped Variable name. In run-mode this reference is replaced by the value of the corresponding variable.

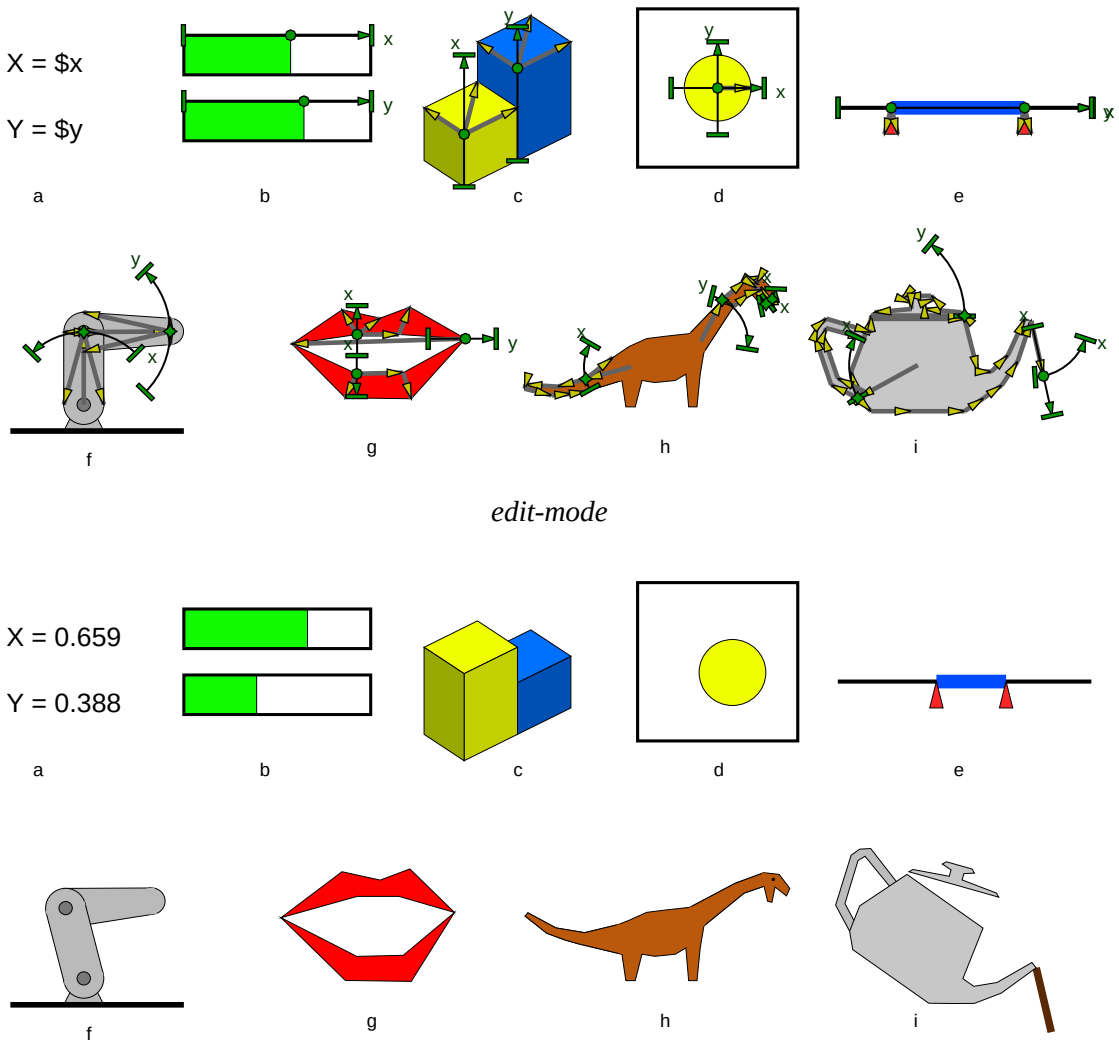

Figure 5: Nine representations of two scalar variables

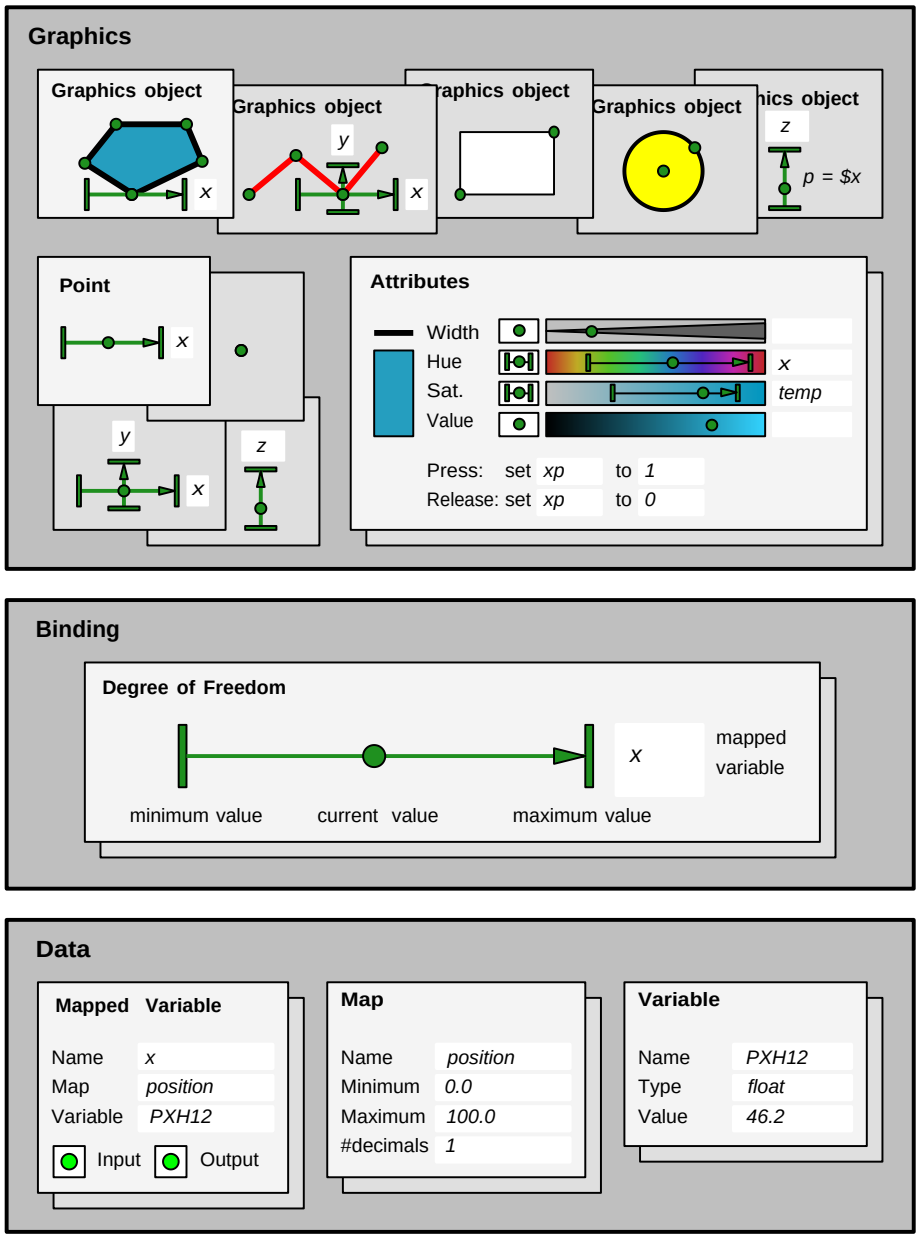

Figure 6: Objects in the PGO editor

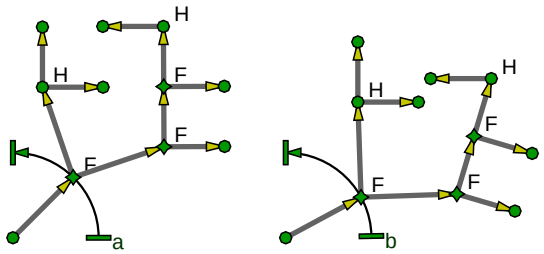

Figure 7: Points and point structures

3.3 Degrees of Freedom

Degrees of Freedom (DOFs) are used to parametrize points and attributes as a function of variables. DOFs have a standard visual representation which is shown in figure 6. Each DOF has a minimum, a maximum, a current value, and possibly a Mapped Variable that is bound to the DOF . In edit-mode all aspects of the DOFs can be changed interactively via dragging and text-editing, in run-mode only the current value can be changed. Seven options are available for assigning DOFs to a point: no DOFs , Cartesian DOFs in either x, y, or both, and polar DOFs in *radius*, *angle* or both (see figure 8). Polar DOFs can only be used for points with parent-points.

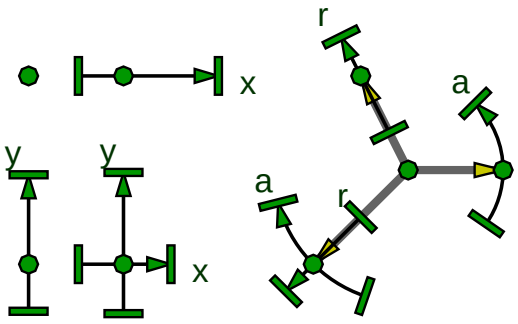

Figure 8: Degrees of freedom for points

With these options in combination with the relations between points a wide variety of coordinate frames can be defined: relative, absolute, Cartesian, polar, hierarchical, although the concept of a coordinate frame itself is not provided. This is a typical example of the main principle that has guided us: provide a set of simple primitives and useful operations to combine them, so that the user can easily construct a wide range of higher order concepts.

DOFs can also be used for the attributes hue, saturation, value, and linewidth, in the same way as DOFs were used to associate a mapped variable with geometry. These attributes are presented to the researcher via a separate attribute window (figure 6). With those DOFs at most four variable values can be visualized simultaneously.

3.4 Mapped Variables

The name that a researcher can specify for a DOF refers to a Mapped Variable (MVAR). The data of a MVAR are references to a Map and a Variable, and two on/off switches for input and output. With these switches the researcher can select if input information (from the researcher to the Data Manager) and output information (from the Data Manager to the researcher) can pass or not.

The Map contains a specification how values must be mapped in the communication with the Data Manager. The current implementation is simple: only linear mappings are supported, hence the specification of a minimum and maximum value suffices. Furthermore, the Map contains a specification of the format for textual output. A Variable is a local copy of the information in the Data Manager. Here some bookkeeping information, such as the type, size, name and Data Manager id, and the current value(s) are stored.

A simpler implementation would be to use just one object that includes all information. However, the indirection via the MVARs is very useful. Several Variables can share the same Maps, thus, to change the mapping of those variables only a single map has to be changed. As an example, suppose that a simulation calculates a variety of water temperatures ($T1$, $T2$, ...), all within the same range (say, 0 to 100 ° C). Here we can use the same map ($temperature$) for all variables. Also, one variable can have several associated mappings. For instance, a wide mapping to select a variety of values, and a narrow one to visualize if a value is above or below a threshold value. As an example, to visualize if water is boiling or not, we can use an additional map $boiltemperature$ from 99.5 ° to 100.5 ° Celcius.

3.5 Output

In the previous sections we have described the objects in the system, now we can describe how they interact in run-mode. We start with visualization: the route from data in the Data Manager to images and animations.

Upon switching from edit to run-mode the PGO editor first makes a connection to the Data Manager. For each MVAR that is used by a DOF (or that is referenced in a text object), and for which the switch *output* is on, the PGO editor expresses interest in mutation events for the associated variables. The algorithm for updating the image is as follows:

1. When a notification event arrives for a variable, its value (say f) is sampled and stored locally.

2. Via the map (say m) a normalized value f_n is calculated:

$$f_n = \begin{cases} 0 & f \leq m_{min} \\ (f - m_{min})/(m_{max} - m_{min}) & m_{min} < f \leq m_{max} \\ 1 & f > m_{max} \end{cases}$$

   If f is outside the range of the map, f_n is clamped.

3. For all DOFs bound to MVARs the current value d_c is updated via:

$$d_c = (1 - f_n)d_{min} + f_n d_{max}.$$

4. The corresponding objects are updated. Colors are recalculated: mapped from HSV to RGB-space. The coordinates of points and all their children are updated according to the type of DOF .

5. Finally, the graphics objects are redrawn, using the changed points and the new values for attributes, and thereby displaying the changes in the data. References to MVARs in text objects are replaced by the formatted value of the variable.

Redrawing the image for each change of the data of the Data Manager is expensive, and can lead to erroneous animations. As an example, consider an object with coordinates x and y. Typically a simulation will change both x and y in a single time-step. If the image is redrawn for each change in the value of one of the variables, the observed trajectory of the object will have a staircase form. Therefore, the researcher can specify a trigger variable. If a trigger variable (say t) has been specified, then the PGO editor only awaits changes in t. When t changes, all relevant variables in the Data Manager are sampled, and the image is redrawn.

3.6 Input

The processing of user input in run-mode is very similar to the handling of output, except that the direction of the data flow is reversed. Points can be dragged, but now, in contrast to edit-mode, only along the direction(s) specified by the associated DOFs . Changes in the corresponding MVARs are dealt with in the reverse way as described in the previous subsection. Child points move along just as they would in edit-mode: the DOFs move along with the point.

The researcher can also change attribute-values of graphics objects in the attribute window, provided that DOFs have been defined for them. This is done by dragging the current value of the DOF in the attribute window to a new value.

Picking is a useful interaction primitive for the selection of objects and for invoking actions. We have defined picking in line with the data-driven concept of the system: a pick results in a change of data. One standard action is predefined: Set the value of a *variable* to a *value*. For each graphics object the researcher can specify which variable has to be assigned what value, both for press-events (a mouse-button is pressed with the pointer inside the graphics objects) and for release-events (a mouse-button is released).

Text objects can be changed in run-mode by the researcher if the text object contains one or more references to MVARs , and if the 'input'-switches of the MVARs are on. The researcher can click at such an object, the string is replaced by an input box, and the researcher can enter new text. Next this text is scanned, the value of the variable associated with the MVAR is updated, and this new value is sent to the Data Manager. Also, the image is redrawn to show updates in other graphics objects that depend on the same variable.

3.7 Arrays

Often the major part of data to be visualized will be arrays. The manual specification of a large number of similar PGOs , with DOFs parametrized as indexed elements of the array, is a tedious process. Therefore, we automated it. Each graphics object in edit-mode is considered as a template, from which instances are generated after binding with the Data Manager. From all graphics objects multiple instances can be generated. In addition, polylines and fill-areas points can be expanded in-line within a single object (see figure 9).

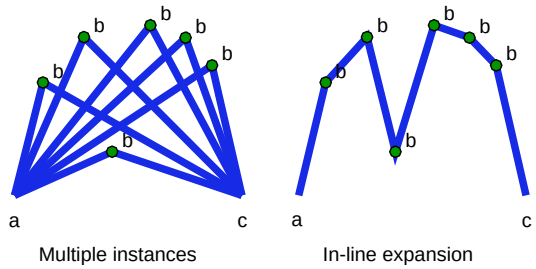

Figure 9: Two different expansions of arrays for polyline object

4 Applications

4.1 Balls

In figure 10 an edit-mode version of figure 2 is shown. We used array-expansion for the balls. Only a single circle with a single velocity-line is defined. Each of the variables px, py, vx, and vy is an array. The mapped variable rc is bound to the variable r, with the input-switch to off. Thus, when a ball is dragged, the radius is not influenced. The variable r is scalar. If r were an array (each ball has a different size), still the same interface could be used. The color of the enclosing box is parametrized to the damping factor of the velocity.

The standard satellite *dmlog* in combination with array-expansion was used to make the graph of the total kinetic energy in the lower-left corner. A polyline for a single point with a suitable parametrization was defined. If logging is active, the point is expanded in a series of points, and hence a sliding graph is displayed. This example shows that standard scientific graphics primitives can easily be defined.

4.2 Cars

Figure 11 shows that the CSE can be used for multi-dimensional visualization, and as a front-end to databases. We used our CSE to visualize a table of 400 different types of cars, where for each car a number of attributes such as displacement, horsepower, and acceleration are given [1]. We developed a small satellite that reads in the data, writes these data to the Data Manager, and that can be used to make selections of the data. The data are visualized via three scatter-plots. Each item in the scatter-plot represents a car type. For each item we used color to visualize the country of origin, and two lines to indicate miles per gallon and weight. Each axis of the scatter-plots has an associated range-slider to make selections of the cars. Further, the user can click at each item, and the properties of this car type are shown in the lower right corner.

[1] The data set has been provided by the Committee on Statistical Graphics of the American Statistical Association for its Second (1983) Exposition of Statistical Graphics Technology.

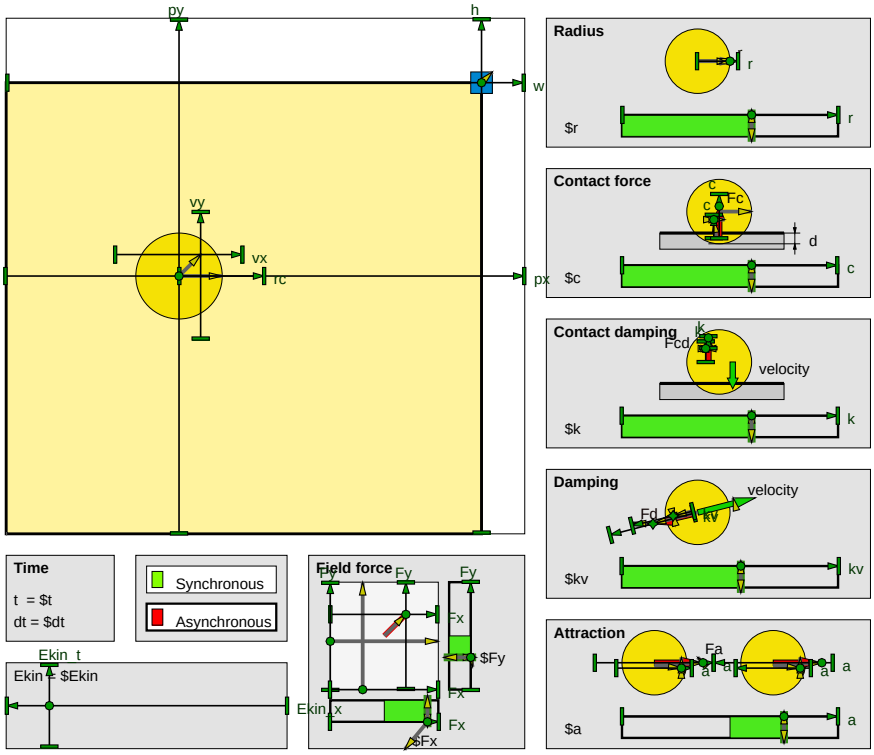

Figure 10: Simulation of bouncing balls, edit-mode

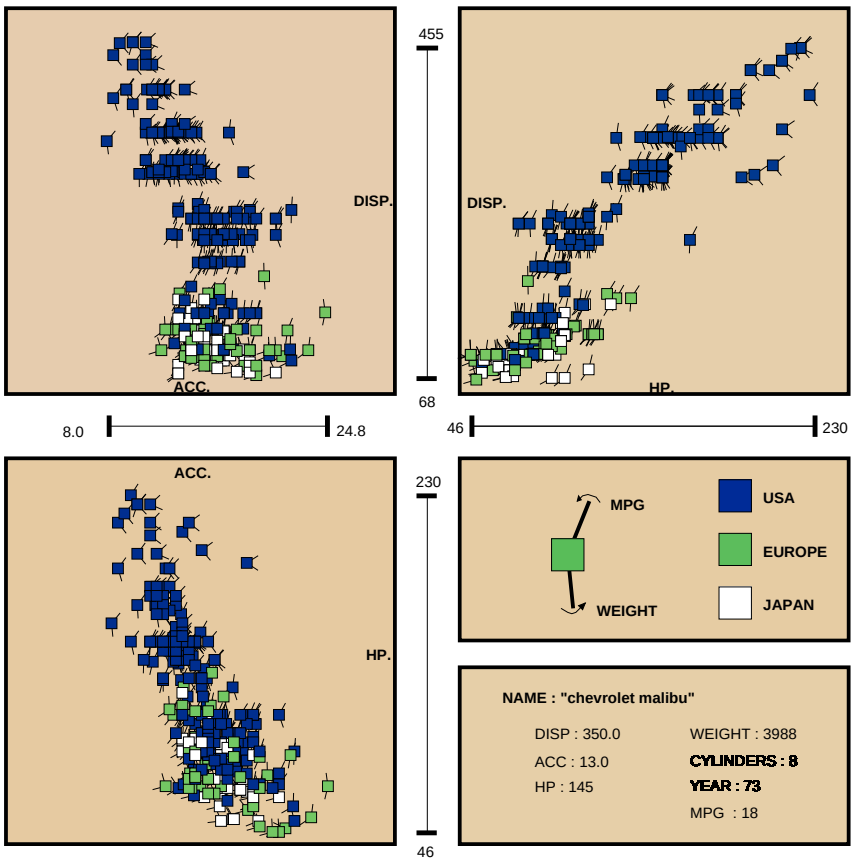

Figure 11: Visual front-end to a multi-dimensional database.

4.3 Atmospheric simulation

We have applied our techniques to a model for smog prediction over Europe. The full blown model forecasts the levels of air pollution, which is characterized by approximately 104 reactions between ca. 70 species. For example, the concentrations of ozone (O_3), sulphur dioxide (SO_2) and sulphate aerosol (SO_4) are calculated. The verical stratification is modeled by four layers; the surface layer, the mixing layer, the reservoir layer and the upper layer. The physical and chemical model is described by a set of partial differential equations that describe advection, diffusion, emission, wet and dry deposition, fumigation and chemical reactions.

An important numerical utility to solve these equations is local grid refinement. This technique is used to improve the quality of the model calculations in areas with large spatial gradients (for example in regions with strong emissions) and in areas of interest , specified by the researcher. The tradeoff to be made in local grid refinement is calculation accuracy versus computation speed.

We have used our environment to steer various aspects of the local grid refinement technique. We name only a few of these aspects:

- the researcher may select the species and the layer to which grid refinement will be applied.
- define a the tolerance value which determines where refinement is necessary.
- define a bounding box as a region of interest to drive the grid refinement.
- interactive control over simulation time.

Figure 12 shows a snapshot of the ongoing simulation. On the left, in the PGO editor, only the refined grid within a region of interest is shown. Visualized is one species (in this case, ozone) at one layer and the wind field. On the right, in the IRIS Explorer Render module, all four layers of the same species is shown. We used the satellite $ReadDM$ to send data to IRIS Explorer. The $dmSlice$ satellite is shown on the lower right.

The amount of data produced per time step of the simulation is substantial. For each grid point the concentration of each species is calculated. Depending on the grid refinement level, the amount of data can vary between 1 to 4 megabytes per time step. There are approximately 500 time steps in our test simulations.

The geometry of the grid-cells and all concentrations are stored as variables in the Data Manager. The *dmslice* satellite is used to interactively select a particular concentration. Concentrations are visualized via specification of a single rectangle that is parametrized with these variables. The wind field is visualized via specification of a single polyline. Both the rectangle and polyline specifications are expanded through the array expansion mechanism.

5 Discussion

In the previous section we have presented applications. In this section we step back to a higher level of abstraction. We discuss the underlying concepts of the developed CSE , compare our work with related work, consider future extensions and the scope of the methods and techniques described.

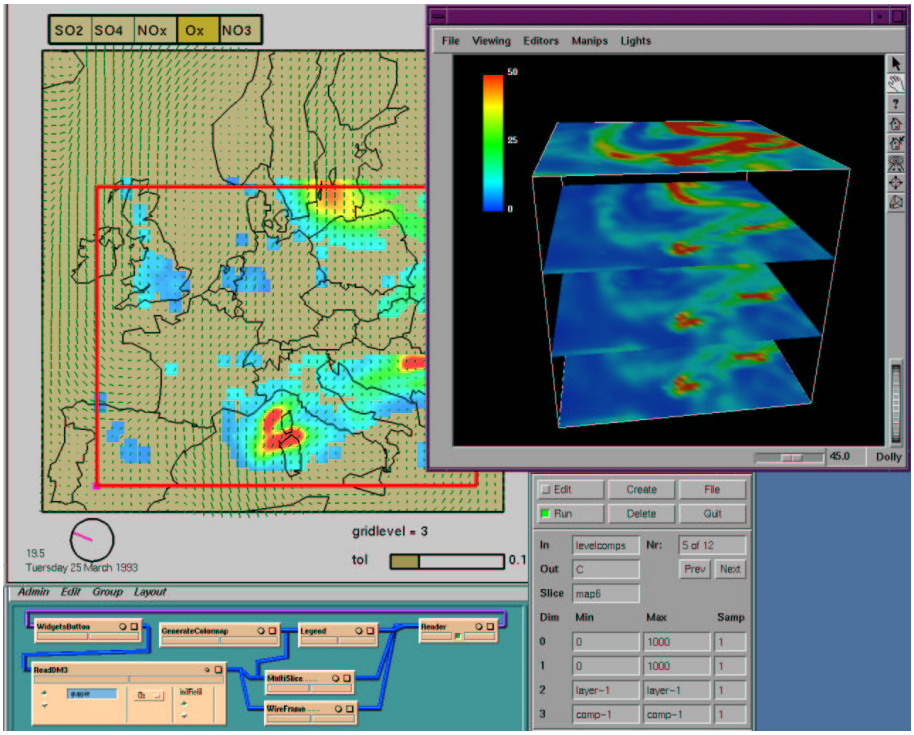

Figure 12: Visualization of a smog prediction model with the PGO editor and IRIS Explorer

5.1 Concepts

The presented CSE has two major components: the Data Manager and the PGO -editor. Several concepts are shared by both components:

- The use of *low-level primitives*: a simple data model and graphics objects. The interfaces to these primitives are familiar to the end-user: a simple I/O library for data manipulation and a graphics editor for graphics.

- *No higher level semantics* are defined for the graphics and the data. As a result, the environment is general and flexible. The researcher can easily add meaning to the graphics and data. With the PGO editor the end-user can edit and combine PGOs to realize meaningful widgets. Changes to existing simulation software are minimal. The locations where hook functions must be added are easy to locate, data structures and program structures do not have to be changed. Summarizing: the CSE itself is not a tool for, say, physically based modeling and animation, but it could be a great environment for the development of such a system.

- Both the Data Manager and the PGO editor rely on *late binding* of names. As a result, it is possible to define new visualizations of the data output by the simulation, while the simulation continues to run.

- All operations in both the Data Manager and the PGO editor are based entirely on *data*. Dragging, picking and text input are translated into changes of data. The main type of event within the system is the data mutation. Process control (firing algorithms or execution models) can be implemented on top of those events by the satellites.

5.2 Comparison

In section 1.3 we considered three approaches to realize an environment for computational steering. Our approach falls in the class of extension of graphics tool-kits. Most related to our work are therefore the Inventor-system and the 3-D interactive system being developed at Brown University. However, there are also important differences. An obvious difference is that those two systems are 3-D. We will discuss the extension of our system from 2-D to 3-D in the next section.

Our starting point was the researcher as an end-user: how can we offer an environment that suits his needs and is easy to use ? This is reflected in the contents of the paper. We extensively discussed how the environment looks from the end-user point of view, and paid less attention to the computer science methods and techniques used. This was a deliberate choice: for us this was the most important issue. The approach taken by Inventor and at Brown University is primarily of interest for application programmers and not for end-users.

Another interesting difference is the handling of user-input and hence, direct manipulation. In both Inventor and at Brown University separate objects are used for this: a distinction is made between geometric objects and manipulators. Relations between those two categories must be specified via text. For the PGOs this distinction was not made. In their terms: each graphics object can serve as a manipulator. Feedback via the manipulated object is always provided. If such a manipulation must have an effect on other objects, then the use of the same names in DOFs suffices for linear mappings. If more complex behavior is required, this can be added as a separate satellite.

5.3 Future work

Concerning the Data Manager, we will consider the use of shared memory and point-to-point communication schemes for more efficient interprocess communication. In the current implementation a write will always transport data to the Data Manager which, in turn, will transport the data to all subscribing satellites. This indirect method of communication can be prohibitively expensive for large data sets. This problem in the current implementation will be overcome by allowing point-to-point channels between satellites. In this way, the Data Manager will act only as a router for large data-sets.

In the current environment all graphics are two-dimensional. This allowed a rapid development of the environment, and an early validation of the underlying concepts. Two-dimensional PGOs do not render the CSE useless. Many real-time simulations are two- or even one-dimensional, so that a 2-D PGO editor is more than justified. Further, the results of three-dimensional simulations can be mapped on two-dimensional data (slices), or data of lower dimensionality. Finally, we have shown that we can include existing tools for 3-D visualization in our environment. Nevertheless, a 3-D version of the PGOs would be welcome. This would mean that we have to implement solutions for viewing, direct manipulation and cursor positioning in 3-D. But that is primarily a problem related to 3-D, and not to our environment. We expect that the basic concepts, and especially the one-dimensional DOFs , can be transposed to 3-D without major problems.

We have tailored our environment to Computational Steering (see figure 1). It is an interesting exercise to replace *researcher* by *user* and *simulation* by *application*. We expect that the environment and its underlying concepts will be useful for many other application areas. We hope to investigate this in the future. Some examples of possible applications of our visual specification techniques: custom widgets in User Interface Management Systems; interactive animation and simulation for Computer Aided Education and hypermedia; and template-driven graphics editors.

6 Conclusions

We have shown that the combined use of:

- an architecture with a central Data Manager;
- parametrized graphics objects;

results in an environment that provides

- easy integration of different processes;
- easy definition of visualizations;
- easy definition of customized user-interface widgets;
- direct manipulation by default.

The primary target application for the environment is computational steering, but the same environment, and at least the same concepts, can probably be applied in many other application areas. In short, we have developed a system that makes it easy for an end-user to specify and use interactive computer graphics.

References

[1] B. McCormick, T. Defanti, and M. Brown. Visualization in Scientific Computing. *Computer Graphics*, 22(6 (SIGGRAPH '88)):103–111, 1987.

[2] R.E. Marshall, J.L. Kempf, D. Scott Dyer, and C-C Yen. Visualization Methods and Simulation Steering a 3D Turbulence Model of Lake Erie. *1990 Symp. on Interactive 3D Graphics, Computer Graphics*, 24(2):89–97, 1990.

[3] P.S. Strauss and R. Carey. An Object-Oriented 3D Graphics Toolkit. *Computer Graphics*, 26(6 (SIGGRAPH '92)):341–351, 1992.

[4] A. van Dam. VR as a Forcing Function: Software Implications of a New Paradigm. *IEEE Virtual Reality Symposium Proceedings*, pages 6–9, 1993.

[5] R. Zeleznik, D. Brookshire Conner, M. Wloka, D. Aliaga, N. Huang, P. Hubbard, B. Knep, H. Kaufman, J. Hughes, and A. van Dam. An Object-Oriented Framework for the Integration of Interactive Animation Techniques. *Computer Graphics*, 25(4 (SIGGRAPH '91)):105–111, 1991.

[6] S. Snibbe, K. Herndon, D. Robbins, D. Brookshire Conner, and A. van Dam. Using Deformations to Explore 3D Widget Design. *Computer Graphics*, 26(6 (SIGGRAPH '92)):351–353, 1992.

[7] B. Myers. User Interface Tools: Introduction and Survey. *IEEE Software*, 6(1):15–23, 1989.

[8] D. Olsen Jr. Workshop on Software Tools for User Interface Management. *Computer Graphics*, 21(2):71–147, 1987.

[9] G. Pfaff, editor. *User Interface Management Systems*. Springer Verlag, 1985.

[10] B. Myers. Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints. *ACM Transactions on Programming Languages and Systems*, 12(2):143–177, 1990.

[11] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schelgel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System : A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(7):30–42, 1989.

[12] C. Upson. Volumetric Visualization Techniques. In D.F. Rogers and R.A Earnshaw, editors, *State of the Art in Computer Graphics*, pages 313–350. Spriger Verlag, 1991.

[13] W. Felger and F. Schroder. The Visualization Input Pipeline – Enabling Semantic Interaction in Scientific Visualization. *Computer Graphics Forum*, II(3):139–152, 1992.

[14] L. Bergman, J. Richardson, D. Richardson, and F. Brooks Jr. VIEW – An Exploratory Molecular Visualization System with User-Definable Interaction Sequences. *Computer Graphics*, 27(6 (SIGGRAPH '93)):117–126, 1993.

[15] J.J. van Wijk and R. van Liere. An Environment for Computational Steering. Technical Report CS-R9450, Centre for Mathematics and Informatics, Amsterdam, The Netherlands, 1993.

[16] IRIS Explorer Development Team. IRIS Explorer ^{TM} Module Writer's Guide. Technical Report 007-1369-020, Silicon Graphics Computer Systems, Mtn. View, CA., 1993.

[17] C. Williams, J. Rasure, and C. Hansen. The State of the Art of Visual Languages for Visualization. In *Proceedings Visualization '92*, pages 202–209, 1992.