Experiences with the Pavane Program Visualization Environment

Authors: Kenneth C. Cox and Gruia-Catalin Roman

The Pavane system is an implementation of the declarative visualization model, which treats visualization as a mapping form the state of some entity (a program, an execution, a data set, etc.) to images. We briefly discuss the declarative visualization model and the implementation of Pavane. We then present some of the results that we have obtained with Pavane. These results are divided into three parts: a discussion of the development of visualizations focusing on the roles played by the various participants, a critique of declarative visualization in general and Pavane in particular, and a presentation of a number of common techniques in the development and construction of visualizations. The last is organized as a series of vignettes appended to the text.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research
Experiences with the Pavane
Program Visualization Environment

Kenneth C. Cox
Gruia-Catalin Roman

WUCS-92-40

October 1992

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899
Abstract

The Pavane system is an implementation of the declarative visualization model, which treats visualization as a mapping from the state of some entity (a program, an execution, a data set, etc.) to images. We briefly discuss the declarative visualization model and the implementation of Pavane. We then present some of the results that we have obtained with Pavane. These results are divided into three parts: a discussion of the development of visualizations focusing on the roles played by the various participants, a critique of declarative visualization in general and Pavane in particular, and a presentation of a number of common techniques in the development and construction of visualizations. The last is organized as a series of vignettes appended to the text.
Correspondence: All communications regarding this paper should be addressed to

Dr. Gruia-Catalin Roman
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

office: (314) 935-6190
secretary: (314) 935-6160
fax: (314) 935-7302

roman@cs.wustl.edu
1. Introduction

Visualization — the graphical display of information — continues to gain in importance in a wide variety of fields. The visual display of large volumes of data has become an accepted, even essential, part of science and engineering and is rapidly becoming vital in other fields such as medicine and management. This is largely due to two factors. First, and most vital, human beings are very good at processing visual information; many of our mental models of the world are expressed in images and diagrams. Second, the performance of graphics workstations is increasing even as the price continues to fall, increasing the accessibility of visual displays.

Our own area of interest is program visualization, which we define as the graphical presentation, monitoring, and exploration of computations. Our particular research focuses more strongly on the visualization of concurrent computations, that is, computations involving a large number of cooperating processes; we are especially interested in the use of visualization as a tool for explaining the behavior of such computations. To facilitate our research, we have developed a program visualization system called Pavane. An earlier version of this system has been described elsewhere [10]. In part, this paper updates that description, but its major purposes are to discuss in general terms the process of constructing a visualization and to present a number of visualizations using the Pavane system.

We choose to consider program visualization as a mapping from some aspect of a program (or execution of a program) to a final image. This treatment leads to a very natural approach to the categorization of program visualization systems, as we have shown in [9]. Fundamentally, a mapping can be characterized by its domain, its range, and by the manner in which the mapping is specified. In direct correspondence with this, we can classify program visualization systems by which aspects of the computation are examined (the domain), by what graphical objects and techniques are provided (the range), and by how the transformation of the program aspects to graphical form is expressed and implemented (the mapping specification).

This model of program visualization as mapping suggests a natural implementation of a visualization consisting of three "computations" working concurrently. The first of these computations is the program whose behavior is to be visualized, or the underlying computation. The second, visualization computation implements the mapping by examining the state of the underlying computation and transforming it into some graphical form. Finally, the rendering computation displays the images generated by the visualization computation, possibly
providing some simple viewer interactions. The word “computation” is used in its most general sense here, to encompass all the many ways in which these functions can be performed. All three computations may be part of a single process, or each computation might itself consist of a large number of closely- or loosely-coupled processes, or even be partly implemented in hardware (particularly in the rendering component).

This tripartite division is also reflected in the roles involved in a visualization. The programmer develops the program, the animator constructs the visualization, and the viewer examines the final images. Ideally, these three roles should overlap minimally. The programmer should certainly be able to construct a program without concern for how it is to be visualized, and the viewer should be able to understand the images without detailed knowledge of the underlying computation or the visualization process. The animator has a more difficult task, for she must work with the structures used by the programmer and produce a result that can be appreciated by the viewer.

All existing program visualization systems can be viewed as implementations of this model, in that all such systems transform some information about the program or its execution to graphical form. In practice, most systems do not explicitly implement a mapping. Rather, the animator identifies those computational events which are of interest (e.g., the exchange of two elements in a sorting algorithm). The animator then modifies the underlying computation, annotating it with procedure calls at those places where these events occur. During program execution, the annotations produce an event trace which is used to generate the final images. The Balsa system [1] and its successor Zeus [2], probably the most well-known and influential algorithm animation systems, use this technique, which we call imperative visualization. In effect, imperative visualizations are mappings in which the domain consists of program events.

As we have argued before [8], the need to identify events and annotate the corresponding points in the code is one of the most significant limitations of imperative visualization. This is particularly true when dealing with concurrent computations, our own area of interest, where an “event” may be a nebulous entity defined by state changes in a large number of distinct processes. In such problem domains, declarative visualization appears to be a more useful and adaptable approach.

Declarative visualization systems are closely related to our basic mapping model, in that the visualization is specified and explicitly implemented as a mapping from the underlying computation’s state to the final image. The animator determines which conditions of the underlying state are of interest and constructs a mapping which detects
these conditions and transforms them into visual form. The visualization computation is assumed to have complete access to the state; in practice, this may require the annotation of the underlying program to transmit the state to the visualization computation. However, since the entire state is examined (rather than animator-defined events), the annotation process could in principle be largely automated and incorporated into compilation.

The remainder of this paper consists of three sections. In the first section, we provide a brief introduction to the Pavane system. This includes both a conceptual overview and a discussion of Pavane's implementation. The remaining two sections describe some of the results of our research with Pavane. We first discuss the roles of the various participants in the construction and use of a visualization; although directed toward Pavane, our remarks are applicable in general. We then evaluate declarative visualization and Pavane, suggesting some further avenues of research. In addition to these formal sections, we have included a series of vignettes which illustrate a number of common techniques used in the construction of visualizations.

2. Pavane

Pavane uses the declarative model of visualization. In Pavane, visualizations of computations are implemented as mappings from the state of the program to a collection of graphical objects in a four-dimensional space — the three spatial dimensions plus time. For notational convenience, we express both the underlying computation's state and the final four-dimensional graphical objects as collections of tuples, called respectively the state space and animation space of the visualization. The overall mapping from state space to animation space can be decomposed into a pipeline of any number of sub-mappings, with each intermediate mapping transforming one space into the next in the pipeline; each of these intermediate spaces is also a collection of tuples. Figure 1 illustrates the structure of a Pavane visualization. We assume that the underlying computation progresses by means of a series of atomic transitions which modify the state. After each transition, the visualization rules are re-applied to the new state and the resulting animation space is rendered.

The reader is referred to our earlier paper [10] for details of the Pavane notation and semantics; we give only an overview here. A tuple is a structured data type having the form \( \text{typename}(\text{component}_1, \ldots, \text{component}_n) \). A space is a set of tuples; each typename belongs to one and only one space. As mentioned previously, the overall mapping from the underlying computation's state space to the animation space can be decomposed into several sub-
mappings; each sub-mapping has an input space and an output space. Each mapping consists of a collection of rules having the form

\[ \text{rulename} = \text{variables} : \text{predicate} \Rightarrow \text{production} \]

The predicate portion of the rule may contain arbitrary simple predicates (such as comparison of variables) as well as tests for the presence or absence of tuples. The rule may examine its input space, the previous instance of its input space (i.e., that resulting from the most recent application of the mapping), and the previous instance of its output space; we have found this one-step "memory" feature to be quite powerful and even essential to a number of common visualization techniques, especially the production of animations. The production part of the rule is a list of tuples in the output space of the rule. Both the predicate and the production may make use of the rule's variables. The overall interpretation of such a rule is, "For every instantiation of the variables such that the predicate is true, the list of tuples in the production is to be placed in the output space."

Pavane's graphical model provides one or more three-dimensional "worlds", each containing a collection of graphical objects. The animator defines one "window" for each world; the window definition includes the world's properties (center, scaling, background color, and so forth), the properties of the screen window which will be opened
(dimensions, position, etc.), and the types of transformations that the viewer is permitted to make (e.g., the viewer might only be permitted to view the world from a point on its Z-axis). Subject to any such limitations that the animator requires, the viewer can examine each world "through" its window from any point in the world's coordinate system. The ability to use multiple windows is especially convenient when the effectiveness of two or more visualizations is being compared; we need only place the visualizations in two windows positioned next to one another.

The animation space consists of tuples which define graphical objects. These tuples have the form

\[ \text{typename}(\text{attribute}_1 := \text{value}_1, \ldots, \text{attribute}_n := \text{value}_n) \]

where the \text{typename} is the graphical object's basic shape (e.g., line, rectangle, sphere) and each \text{attribute}/\text{value} pair is the name of some property of the object (window, color, position, radius, etc.) and the value that is to be assigned to that attribute. Attributes which are not included are assigned default values. Attributes may vary in time; this is accomplished by making the associated value a function of time, which is measured in terms of frames (single images — an animated graphical transition is a sequence of frames). As an example, the following tuple generates a sphere that smoothly moves from coordinate \([0,0,0]\) to coordinate \([20,20,-10]\) between frames 5 and 15:

\[
\text{sphere}(\ \text{radius} := 1.0, \ \text{center} := \text{ramp}(5, [0,0,0], 15, [20,20,-10]))
\]

Pavane's implementation has changed greatly since our previous description [10]. The system described in that paper was largely Prolog-based. The underlying and visualization computations were implemented as a single process, the Prolog interpreter. Prolog goals representing the underlying computation (translated from the Swarm language [5]) and the visualization mapping (translated from Pavane's rule-based notation) were loaded into this process and interpreted; the results were sent to the rendering process for viewing.

Prolog provided a number of advantages — the visualization rules could be modified and re-loaded rapidly, and new data types and constructs were relatively simple to add — but the resulting computation was terribly slow due to the multiple levels of interpretation involved. As our visualizations grew more complex, this became first an annoyance and then an active hindrance to our research. We briefly considered using a Prolog compiler, but rejected the idea in favor of a complete move to another compiled language. We chose C for reasons of portability and familiarity, and re-built the Pavane system accordingly.
Pavane now consists of five components, as shown in Figure 2. The program sc (for "Swarm compiler") translates Swarm programs into C code. This code is then compiled and linked with SwarmLib, the Swarm run-time execution library, to produce an executable file for an underlying computation. As an alternative to the use of Swarm, a C program may be linked with the CioVis library to produce a underlying computation. To use this library, the animator must insert function calls into his code to indicate that particular variables in the C code are to be monitored. Additional calls are added to indicate significant state transitions.

Visualization rules are similarly compiled by sc and linked with the VisLib library to produce an executable visualization computation. Finally, the rendering computation, called simply Display, is little changed from the version described in our earlier paper. We now have two versions of Display, the first using the Silicon Graphics Personal Iris™ gl library with its high-quality graphics and the second using the more portable, but also more primitive, X Window System® graphics.

A visualization is executed by running the underlying computation with arguments which indicate which visualization and display programs should be used. The initiation of these processes and the establishment of communications links between them is then automatic. As the underlying computation runs, each state change
(execution of a Swarm synchronous group in a Swarm program, or reaching of an animator-defined checkpoint in a C program) is transmitted to the visualization computation and mapped to an animation space which is in turn transmitted to the rendering computation and displayed. Actually, a visualization may involve several rendering computations — each window defined by the animator is controlled by a separate rendering computation. This opens up interesting possibilities for visualizations whose output is sent to different screens (possibly on distant machines) which we have not yet exploited.

The overall process of visualization is data-driven; the rendering computation must wait for data from the visualization computation, which in turn waits for data from the underlying computation. It is not, however, synchronous. The underlying computation is permitted to send state changes more rapidly than the visualization computation is able to process them, and similarly the visualization can send animation spaces faster than the renderer can translate them into images. The latter often happens, simply because the process of generating an image (and, more importantly, the comprehension of that image by the viewer) is much slower than the other two computations. If synchronous execution is desired, it can be requested when the computations are started; the underlying computation is then forced to wait for the rendering computation to complete the display of an animation space. This is particularly useful when contrasting two or more different visualizations by displaying them in separate windows, since synchronization guarantees that all the images represent the same state of the underlying computation.

As expected, the move to C had both disadvantages and advantages. Because visualization rules are no longer interpreted (in fact, they are compiled twice, once by sc and once by CC), we are no longer able to modify the rules "on the fly" — that is, introduce a new set of visualization rules into an executing computation and observe the result. Instead, the entire visualization must be halted and a new one started. This minor drawback is more than offset by the increase in execution speed (our early tests gave a speed-up factor of as much as 100 for such common operations as finding a tuple) and in portability.

3. The Participants' Roles

Each of the three sub-sections below describes the process of visualization as seen by each of the three participants — the programmer who develops the underlying computation, the animator who creates the
visualization, and the viewer who examines the final images. In each case we present some general remarks, then apply them to Pavane.

The programmer's role

The programmer develops the underlying computation, that is, the program whose state is examined during the presentation of a visualization. Ideally, we want the programmer to have no role in the construction of a visualization. That is, the programmer should be able to program in a normal style, unconstrained by the fact that the code will be visualized. This goal is largely achieved in most visualization systems; even in systems where the code must be modified (e.g., through the addition of procedure calls marking key events), this task falls to the animator, who must insert the appropriate annotations in the existing code.

The programmer does have one more role to play after he has produced a complete program. The animator's job is to transform information about the program into visual form; in order to do this, the animator must understand the structure of the program, particularly the data representations chosen by the programmer. The programmer must therefore either very carefully document the program, or remain "on call" to explain it. The first approach is preferable, and of course should be done in any case. However, experience indicates that visualization is much like software maintenance: It quickly uncovers any errors or limitations in the program's documentation, necessitating the involvement of the original programmer.

In Pavane, the programmer's role takes two paths depending on whether the underlying computation is written in Swarm or in C. In Swarm, once the program is finished the programmer's task is almost complete; the Swarm dataspace completely defines the state space of the computation, while the Swarm execution model defines the atomic state transitions. When C is used, the programmer must perform the previously-mentioned tasks of documenting the program, including explaining his state representation and helping the animator identify those points where a meaningful state transition is complete.

The animator's role

The animator has the most challenging task in the development of a visualization, since she must transform the semantic content of the underlying computation into a readily-understood graphical form. The animator thus has a very difficult task: She must understand the underlying computation well enough to extract the information she requires, while simultaneously viewing the visualizations without the bias of knowledge of the underlying
computation. In addition to these tasks (which combine aspects of the programmer’s and viewer’s roles), the animator must construct and implement the transform between the program and the graphical form — a task which may itself involve significant programming. The use of an appropriate "toolbox" of techniques, heuristics, and subroutines can greatly assist the animator in his task. Several elements from such a toolbox are illustrated in the various vignettes accompanying this paper; we summarize a few of the more useful ones here.

The central goal of any visualization is to convey information. The selection of the information to be conveyed is key to the effectiveness of a visualization. As our own work with the Floyd-Warshall shortest path algorithm (in [4], and also briefly described elsewhere in this paper) indicates, a simple direct presentation of the underlying computation’s state and behavior is often not effective even though all information about the algorithm is represented. In such cases, a more abstract visual representation is required, and heuristics for the selection of such a representation form an essential part of the animator’s toolbox. Our own research has largely focused on the use of program correctness techniques in the selection of visual abstractions. In the Floyd-Warshall example, this led to the identification of a particular invariant as key to understanding the algorithm, which in turn led to the development of several effective visual representations.

An internal representation of the information of interest, that is, a collection of data structures which maintains the information in a form convenient to the animator, is used in almost all visualizations. In a few rare cases the animator may be able to use the programmer’s own structures, converting them directly into a visual form. More typically, the animator will find the programmer’s structures inconvenient for his particular needs and will have to develop his own. Such structures serve both to abstract and project the underlying computation’s data and to store values that the animator finds useful (typically geometric data about the visual representation). In Pavane, such internal representations can be constructed using intermediate spaces in the mapping as illustrated in one of our vignettes. A history one or more levels deep may be kept in such an internal representation, allowing the previous values of some of the data to be inspected. Such an internal representation is almost de rigueur when animation (smooth visual motion) is involved, since the animator must be able to compare the two successive visual states and arrange for the transition between them. Pavane provides a one-level deep history, allowing the animator to examine the previous space, and with this facility an arbitrarily-long history can be easily constructed.
On the program side of the transformation, the key task is the extraction of information from the program. This may be quite simple and direct, as when an array of values will be transformed into an array of objects arranged along a line, or it might be extremely difficult, as when a complex data structure using memory pointers must be analyzed. Several useful techniques apply here. Projection in its most general sense — the discarding of some data — is almost always required to eliminate aspects of the state which are of no interest to the animator. In systems which use annotated procedure calls to pass program information to the visualization, projection is implicit; only information of interest is transmitted. Other systems may make projection more explicitly; for example, in Pavane a tuple containing ten components might be reduced by a rule to one containing three. An incremental approach, when appropriate, is very powerful and convenient; changes in the underlying computation are translated into changes in the animator’s own data structures, without the need to re-construct the entire structure. This is particularly applicable when structures containing memory pointers are involved, since such structures often evolve through the addition and removal of single elements.

Construction of an effective visual representation of data remains as much an art as a science, and almost any approach must be viewed before its suitability can be judged. However, a few general techniques can be identified; Tufte [Tufte, 1991 #542], among others, has listed many. Geometry is obviously a major component of any visual abstraction, since the arrangement and size of objects can be used to represent data in many ways, some of which are shown in our Bagger example. Here Pavane has an advantage over most other systems, in that it provides a three-dimension "world" of objects. Other graphical properties, such as shape, color, and texture, are less appropriate for quantitative data but remain powerful tools for representing relationships. Animation provides a few techniques to convey information, including flashing an object to draw the viewer’s attention. A relatively powerful animation technique is often overlooked because it is so obvious: By smoothly transforming one property of an object, the viewer can readily perceive that other properties of the object do not change.

The viewer’s role

The role of the viewer is similar to that of the programmer, in that ideally her role should be uninfluenced by the internal workings of the visualization. The viewer should simply be presented with some attractive pictures which, when coupled with appropriate explanations, increase her understanding of the underlying computation. In an exact sense, the viewer must act as a critic of the visualization; she should examine it with the question, “Is this
piece really effective at what it is trying to do?” And, like a critic, the viewer must not hesitate to answer “No!” and explain why the visualization fails.

This is where the viewer plays his most significant role, interacting with the animator to perfect the visualization and ensure that it succeeds. The process is iterative, with the viewer noting that some aspects don’t quite work, the animator modifying those portions, and the viewer examining the result and suggesting further modifications. This repetitive process makes rapid prototyping of a visualization very desirable. Pavane’s use of rules facilitates such iteration, since the rules can be readily added, modified, and deleted, and their form (an arbitrary predicate over the input space producing a list of objects in the output space) permits simple specification of relatively complex transformations. In addition, because the effect of one rule is largely independent of the actions of others, the animator and viewer can focus on portions of the visualization by only using an appropriate subset of the rules; this also facilitates prototyping, since the various parts of the visualization can be prepared separately and then combined.

4. Evaluation

Our work with Pavane has confirmed many of our original expectations about the declarative visualization paradigm while leading us to modify others. The most pleasing result was the confirmation that the declarative approach permits rapid prototyping and an experimental approach to the development of visualizations. This means that it is possible to quickly construct multiple visualizations of an algorithm and compare them to select the best. We are also able to iteratively develop a visualization as described in the previous section. By way of anecdotal evidence, we recently constructed six different visualizations of a sorting algorithm (the quicksort), including some that used three dimensions and relatively complex animations. The development of these visualizations required less than three hours, admittedly by a person with considerable experience with the system.

The results of our methodological investigations have not been quite as good but are still encouraging. Our original thesis was that the program correctness properties of an algorithm can be used to guide the development of a visualization, and that the particular form of a property (universally vs. existentially quantified, for example) would further direct the construction of a corresponding rule. The first of these points has been shown true. Correctness properties do provide valuable guidelines for visualization, and we have used correctness properties to construct a number of interesting visualizations that an ad-hoc approach would probably not have discovered. We refer to this
approach as "analytic visualization". Unfortunately, our second hope — that properties could be used to develop specific rules — has not proved correct. Certainly some general heuristics can be applied in the development process, but a formal mapping from properties to rules seems unlikely.

We are also pleased by our efforts to make Pavane available to a wider community. Several groups have expressed an interest in using the system, and several others have actually worked with it. The results from these other groups are good — there seems to be a relatively short learning curve involved, and most of the users have expressed satisfaction with the rule-based notation. The participation of these other groups brought out a number of shortcomings with the earlier implementations, and we have updated the system accordingly (adding, for example, the array data type). These updates have also been accomplished with relative ease, which would seem to indicate that the fundamental system design is sound. New graphical object types, for example, can now be added simply by modifying three tables and adding the code to Display (the rendering computation) to draw the object.

In the interests of honesty, and to indicate further lines of research and development, we must also list some of the problem areas that we have encountered. Fortunately, most of these are concerned with the implementation of the system and not with the underlying model.

The only problem we have identified with the declarative model lies in the area of viewer interaction. It is often desirable to permit the viewer to affect the underlying computation and/or the visualization through the image, for example by "clicking and dragging" one of the graphical objects. Such a change is conceptually difficult to incorporate in the declarative model, since the model treats visualization as a passive process — visualization is an observation of the underlying computation and does not influence its behavior. We have been working with the idea of a "reverse mapping", i.e., a collection of rules which translate the viewer's actions into changes in the underlying computation's state, with somewhat encouraging results.

Among implementation issues, the execution speed of Pavane could be improved. The largest share of execution time is devoted to the evaluation of rules, with repetitive evaluation occupying most of the time — each state change triggers re-evaluation of all the rules. Clearly an incremental approach, in which changes in the state space are used to calculate corresponding changes in the animation space, would be more efficient. The basic problem in the evaluation of a Pavane rule is to find all instantiations of the rule's variables such that the predicate is true. This is very similar to the "many-object many-pattern" pattern-match problem encountered in rule-based
applications, and our research indicates that a variant of an algorithm such as RETE [7] could be used to evaluate the predicates of Pavane rules.

Pavane's mechanism for specifying animations as collections of graphical objects with time-variant attributes is relatively primitive. Several authors have suggested mechanisms which might be more suitable. Constraint-based systems such as ANIMUS [6] specify animations as collections of objects and relationships between the objects. In a constraint-based system, for example, one can specify that several objects are to be positioned at a particular distance and offset from another object, with motions of the latter object immediately causing corresponding motions in the others. TANGO [11] defines animations as a combination of objects and "paths" applied to objects. A path specifies modifications to one or more properties of the object; thus, one can specify a path in a color space which indicates how the color of the object changes.

Finally, some mechanism for generating orchestrated presentations of visualizations would be convenient. At the moment, this is handled by a collection of UNIX® shell scripts, but it is becoming increasingly clear that a better mechanism is needed. This may involve the addition of a fourth role to the visualization process, as Brown and Sedgewick describe [3]. Their scriptwriter prepares presentations of one or more visualizations designed to explain an algorithm, a role which we currently allocate to the animator.

5. Conclusions

The declarative visualization paradigm, as embodied in Pavane, has proven to be an effective means of constructing visualizations. Our investigations have confirmed most of our early expectations for declarative visualization, particularly the hope that the paradigm would support rapid prototyping and visual exploration. This ability to quickly produce high-quality three-dimensional animated visualizations has been of great help to our work on visualization methodologies, to our investigations of visual abstraction, and to our development of a suite of visual techniques as described in the vignettes. We remain convinced that declarative visualization techniques will form an important component of future visualization research.

Acknowledgments

This work was supported in part by the National Science Foundation under the Grant CCR-9015677. The Government has certain rights in this material.
The authors wish to thank Jerome Plun and Don Wilcox for their work on the implementation of Pavane, and Maryellen Saur, Charles Mead, Charles Calkins, and Samudra Sengupta for their development of visualizations.

References


Capturing and maintaining historical information

The visual presentation of many algorithms is enhanced if we present a series of images showing the status of the algorithm at various times. In physical simulations, we often want to record and display the values of one or more variables at regular intervals. A similar problem, often encountered in monitoring, is to present statistics about the behavior of the system, such as a histogram of time delays in a message-passing system.

The common issue in all these applications of visualization is the need to gather and retain the historical information. This data is generally not stored (and indeed, should not be stored) by the underlying computation. The animator must therefore construct and maintain the needed data structures as a part of the visualization computation. Several approaches to this problem are possible. Imperative systems such as BALSA typically utilize some data structures which are accessed and modified by the event-handling code. The declarative approach is similar in principle, differing chiefly in that the animator must define rules to maintain the data.

A simple example of this in Pavane is seen in a visualization of medical data. In this application, we have some instrument which is producing data points at regular intervals, and we wish to display a trace of this data as it might appear on a strip chart. We can model the instrument as a tuple \( \text{data}(\text{value}) \) which changes in each step of the underlying computation. What we wish to produce is a collection of tuples \( \text{sample}(\text{time}, \text{value}) \) which represent the time-stamped successive values — in other words, a history of the \( \text{data} \) tuples.

We can accomplish this with a single mapping, from the state space containing the \( \text{data} \) tuple to a second space (let us call it the \( \text{sampling space} \)) which will accumulate the \( \text{sample} \) tuples. To produce the timestamps, we define a tuple \( \text{timestamp}(\text{time}) \). Initially the tuple \( \text{timestamp}(0) \) is in the sampling space. The rule

\[
\text{UpdateTimestamp} = \text{integer } t : \text{old}.\text{timestamp}(t) \Rightarrow \text{timestamp}(t+1)
\]

creates a new \( \text{timestamp} \) tuple each time. The query part of this rule examines the previous sampling space (as indicated by the \( \text{old} \). prefix) for a \( \text{timestamp}(t) \) tuple, and when it finds such a tuple it generates a new tuple \( \text{timestamp}(t+1) \) in the current sample space. Two rules suffice to generate and maintain the \( \text{sample} \) tuples:

\[
\begin{align*}
\text{GenerateSample} &= \text{integer } t ; \text{ real } v : \text{old}.\text{timestamp}(t), \text{data}(v) \Rightarrow \text{sample}(t,v) \\
\text{MaintainSample} &= \text{integer } t ; \text{ real } v : \text{old}.\text{sample}(t,v) \Rightarrow \text{sample}(t,v)
\end{align*}
\]

The first of these examines the previous sampling space to find the timestamp and the current state space to find the data and creates a corresponding \( \text{sample} \) tuple. The second simply copies all the \( \text{sample} \) tuples from the previous
sampling space to the current sampling space; the same effect could be obtained by declaring *sample* tuples to be *persistent* in the visualization rule file, which indicates that once such a tuple is created, it remains forever.

We can then map the *sample* tuples to a chart, connecting successive samples with lines, by a rule such as:

\[
\text{DrawSamples = } \\
\text{integer t; real v1, v2:} \\
\text{sample(t,v1), sample(t+1,v2)} \\
\Rightarrow \\
\text{line( from := [ t, v1, 0 ], to := [ t+1, v2, 0 ] )}
\]

The image in Figure VA-1 was obtained using essentially these rules; two data sources are monitored, resulting in two graphs, and some additional rules are used to draw the grids.

![Figure VA-1](image-url)
Extracting information from the program

A common problem in the construction of visualizations is the need to derive information which is not explicitly maintained by the underlying computation. This often arises when the animator wishes to emphasize some particular point about the computation's behavior. Perhaps, as in the shortest-path example presented in another vignette, the underlying computation operates on the distance between nodes in a graph but the animator wishes to actually show the paths; or perhaps a simulation maintains position and velocity information but the animator wants to display acceleration. A second area where the animator must synthesize information is commonly encountered when some aspect of the computation must be changed into a geometric form. For example, many algorithms make use of data structures which can be effectively rendered as trees or graphs. However, the underlying computation will almost certainly not include any geometric information in the data structure, and the animator is thus required to generate this information in order to produce a layout of the tree or graph.

We use a distributed priority problem to illustrate the synthesis of geometric information. We are given a directed, acyclic graph. The underlying computation modifies this graph by selecting a node toward which all the arcs shared by that node are directed; when the node is selected, all the arcs are reversed. This behavior is illustrated in Figure VB-1. When two nodes are connected by an arc, we say that the node toward which the arc is directed has priority over the other. Thus, in the left half of Figure VB-1 node 7 has priority over all its neighbors, while node 3 has priority over node 4 but not over nodes 1 or 5. Priority is also transitive, meaning that if a directed path from node $i$ to node $j$ exists, node $j$ has priority over node $i$. Again referring to the left portion of the Figure, node 3 has priority over node 6 because of the path that passes through node 4. It is possible for two nodes to have no relative priority because no path exists between them. Nodes 2 and 4 in the Figure are an example. Because priority is represented by local decisions rather than by a central authority which assigns priorities, we say that priority is implemented in a distributed fashion.

We wish to display the graph with the priority information in three dimensions, using the X and Y dimensions to lay out the graph and using the Z dimension to represent the priorities; specifically, if node $i$ has priority over node $j$, we require node $i$ to have a greater Z-coordinate than node $j$. If the relative priorities of two nodes are indeterminate, such as nodes 2 and 4 in the left half of Figure VB-1, they are permitted to have any relative
heights. A method of converting the distributed, arc-based priority information into the Z-coordinates is not immediately obvious; most schemes would seem to involve some rather complex propagation algorithms to calculate global priorities from the arcs, scarcely the sort of thing one wishes to express in a rule-based form.

However, an alternate approach is possible; this is to update the priorities in an incremental fashion. Imagine that we have, for each node $i$ in the graph, a tuple $\text{priority}(i,v)$ where $v$ is the global priority of the node (the Z-coordinate). Now imagine that node $\text{LastChosen}$ is selected and its arcs are reversed — how should the $\text{priority}$ tuples change? The priorities for nodes other than $\text{LastChosen}$ can be left unchanged, since we can easily show that the reversal of the arcs does not introduce any new paths between such nodes and therefore does not change these nodes’ relative priorities. We can thus simply copy the $\text{priority}$ tuples for these nodes. On the other hand, the priority of $\text{LastChosen}$ must change — it must now be less than that of all its neighbors, since all the arcs from $\text{LastChosen}$ are directed toward these neighbors. We can calculate this new priority by finding the minimum of all the neighbors’ priorities and subtracting 1, as in the following rule:

\[
\text{ChangePriority} = \\
\text{integer mh:} \\
\quad \text{mh} = \left( \min \text{ integer } j, jh : \text{neighbor}(\text{LastChosen},j), \text{old.priority}(j,jh) \mapsto jh \right) - 1 \\
\quad \text{priority}(\text{LastChosen},\text{mh})
\]

This should be read as, "For each value of $mh$ such that $mh$ is equal to the minimum previous priority of any neighbor of $\text{LastChosen}$ minus one, generate a $\text{priority}(\text{LastChosen},mh)$ tuple". Figure VB-2 shows two images generated using this rule, corresponding to the two graphs in Figure VB-1. These images are viewed from the side, with the positive Z-axis directed upward. The node corresponding to $\text{LastChosen}$ (node 6 in the left half, node 7 in the right) is colored green, nodes that have priority over all their neighbors (detected using the predicate $\text{priority}(i,h), (\forall \text{ integer } j, hj : \text{edge}(i,j), \text{priority}(j,hj) : h > hj)$) are colored yellow, and other nodes are colored red. Note how the position of node 7 moves downward when it is selected and its arcs are reversed.
Exploring abstraction

When constructing a visualization, we often want to explore various methods of representing the computation in order to select a clear, effective means of communication. Declarative visualization, with its ability to rapidly create and modify visualization rules, provides an excellent vehicle for such explorations. We illustrate such exploration with the Floyd-Warshall shortest path algorithm. Given an undirected graph with a distance \( d(i,j) \) associated with each edge \((i,j)\) of the graph, this algorithm computes the shortest distance between each pair of nodes in the graph. It uses a single data structure, an array \( d[i,j] \) whose entries are updated in parallel. Throughout the computation, \( d[i,j] \) (if not \( \infty \)) is the distance of the "currently-known" shortest path from node \( i \) to node \( j \). The computation proceeds in a series of parallel steps; in each step, a node is selected and "scanned", which results in the simultaneous update of all the entries of a two-dimensional array \( d \). When the algorithm finishes, \( d[i,j] \) is the shortest distance from node \( i \) to node \( j \).

We might visualize this algorithm by directly representing the array \( d \) — for example, we might write a rule that transforms each non-infinite entry of the array into a box whose height is proportional to the value in the array, and arrange the boxes in a square as suggested by the array. An image from such a visualization is shown in Figure VC-1; the single rule used to construct this visualization also uses color to encode the values of the array. Although most of the computation's state is represented in this array, the visualization does not particularly enhance the viewer's understanding of the algorithm and must therefore be regarded as a failure.

At this point we might decide that our abstraction is too direct; perhaps the visualization will be more successful if we present some other aspects of the state or the computation's behavior. Examining the algorithm, we find the interesting property "At all times \( d[i,j] \) is the length of the currently-known shortest path from \( i \) to \( j \)." This suggests that presentation of the paths would be helpful. Path information is not explicitly maintained by the underlying computation. However, by examining the program's invariants (the logical predicates that describe the program's behavior), we are able to see that an edge \((i,j)\) is on a shortest path from \( h \) to \( j \) precisely when \( d(h,i) + dist(i,j) = d(h,j) \). Developing a rule to detect this and store the edge \((i,j)\) in a pathedge tuple is then trivial:

\[
\text{GetPathEdges} = \text{integer } i,j : d[h,j] \neq \infty, d[h,i] + dist(i,j) = d[h,j] \Rightarrow \text{pathedge}(i,j)
\]
A brief experiment shows that displaying the paths for every pair of nodes overwhelms the viewer, and accordingly we focus on the paths associated with a single node as in Figure VC-2. Again, the distance information is encoded by the colors of the nodes. Viewing this visualization is somewhat more informative — we can now see the shortest paths as they grow through the graph, and observe the corresponding changes in the distances — but something is still missing. We still lack any information about why particular paths are examined and constructed.

The key is once again in the invariant, more precisely in the meaning of “currently-known shortest path”. A more technical statement of the invariant would be, “At all times $d[i,j]$ is the length of the shortest path from $i$ to $j$ all of whose internal nodes have been scanned”. We should therefore indicate the status, scanned or unscanned, of each node in our visualization. A number of abstractions are possible; one such is shown in Figure VC-3. This representation consists of two superposed parts. The first of these is a planar graph showing the underlying graph and, for each node in the graph, its status; scanned nodes are green and unscanned nodes are red. The second part is a three-dimensional collection of spheres representing the shortest path information. The X-Y coordinates and color of the sphere are the same as the corresponding circle of the planar graph; this serves to visually link the two, a link which is reinforced by the thin line connecting the sphere and the circle. The Z-coordinate is proportional to the distance of the sphere from the “focus” node. Since the focus node is at a distance of zero from itself, the corresponding sphere and circle overlap. The focus node is further marked by several circles. This version of the visualization is at last successful, as the viewer can now see the scanning process and observe how the scanning of a node results in the addition of other nodes.

Once we have determined what should be represented — in this case, the shortest paths and the status of each node — we can experiment with different representations. Figures VC-4 and VC-5 are from one such experiment. This visualization completely discards the distance information (the only information presented in our first, direct visualization!) in favor of emphasizing the paths and node status. The underlying graph edges and the unscanned nodes are de-emphasized by putting them in a light color; path edges and scanned nodes stand out boldly. Observing this visualization, particularly when transitions such as that between Figures VC-4 and VC-5 occur, gives the viewer an excellent intuition of how the algorithm functions.
Figure VC-4

Figure VC-5
Visual techniques

The animator's task is facilitated through the use of an extensive palette of techniques for conveying information. We illustrate a few of the techniques using the Bagger program, which implements a parallel best-fit bin-packing algorithm. The computation begins with a collection of "unbagged items" of various "weights". As the computation proceeds, it creates "bags" and places the unbagged items into the bags. Bags have a fixed capacity, and the sum of the weights of the items in the bag cannot exceed this capacity. We have previously presented [10] a visualization of this program; here we discuss three of the visual techniques used in its construction.

The first, relatively trivial technique is that used to convey the idea that items are placed inside bags. In the underlying computation this is represented by setting the item's "bag number" to the identification number of the bag. This suggests a corresponding visual representation in which arrows are drawn from the item objects to the corresponding bag objects; this might be quite effective in another context, but not here. A second representation of the relationship might use color, where the color of an item indicated which bag it was in; in this case there might not be any explicit object corresponding to each bag. Although more suitable than the arrow representation, this approach is also not entirely effective. A third approach, and the one we use, is to illustrate the "insideness" relationship using the position of the objects. This is easily accomplished by representing the bags by a large object and by positioning the smaller "item" objects so that they appear to be inside the bag objects. This is illustrated in Figure VD-1, where the items are represented as colored circles placed in the rectangular regions representing the bags; unbagged items are represented by circles placed to one side of the bags.

One of the properties which we wish to illustrate is that the sum of the weights of the items in a bag does not exceed the capacity of the bag. Here, we wish to illustrate a quantitative relationship and so should use a quantitative geometric property of the objects. Since we have co-opted position for the "insideness" relationship, size is the next obvious choice. We wish to illustrate that the sum of certain numbers is not greater than some other number. This suggests a representation in which one dimension of the object (say, the Y-size) is a linear transform of the value in question (i.e., the item weights or bag capacities). The summation is represented by abutting the item objects along a line so their Y-sizes visually sum. The comparison of this sum to the bag capacity is accomplished by appropriate alignment of the bag object's Y-position with that of the item objects, as shown in
Figure VD-2. This use of dimensions to represent numbers and alignment to represent summation occurs again and again in visualizations, and the interpretation of the resulting image rarely needs to be explained. We have so far been unable to identify a corresponding natural representation for products; this may be because ordinary objects, which essentially define our visual expectations, do not combine in such a way that some property of the combination is the product of the corresponding properties of the constituent parts.

Our third technique involves animation to convey continuity; that is, by showing the transition from one state to another as a smooth interpolation between the states, we reassure the viewer that certain aspects of the state remain unchanged by the transform. Consider such a simple action as placing an item in a bag, which entails the disappearance of the representation of the unbagged item and its reappearance in a bag. If we simply handled it this way — with a disappearance and reappearance — the viewer would not be able to perceive that the item's weight remained unchanged by the movement; indeed, since in this parallel algorithm multiple items may be bagged simultaneously, the viewer may not even perceive that the number of items remains constant. By animating the transition between states, we clearly show the viewer that no untoward changes in the data occur. Figure VD-3 shows a frame from such an animated presentation; naturally, the print medium does not properly convey the effect of the moving image. In this case we have attempted to give a feeling of motion by representing the moving object with multiple graphical objects. A single rule is sufficient to generate all three moving objects in this figure; the three are rectangles, having corner attributes as follows:

\[
\text{ramp(MOVE\_START\_TIME, p1, MOVE\_END\_TIME, p2) - ramp(MOVE\_START\_TIME + DT, p1, MOVE\_END\_TIME + DT, p2) = ramp(MOVE\_START\_TIME + 2*DT, p1, MOVE\_END\_TIME + 2*DT, p2)}
\]

We thus get three rectangles whose corners move from \( p1 \) to \( p2 \), with a time delay of \( DT \) between the movements.
Program development

Visualization can play a significant role in the development of programs, particularly in the detection and correction of programming errors. Visual "surprises" — the appearance of unexpected objects, or the occurrence of incorrect visual events - serves as an immediate notification that something is wrong with the program or with the visualization. A visualization can also hint that there may be something wrong with the underlying computation without a dramatic event; sometimes the images just don't look "right", or expected events do not occur.

We have encountered events of both types many times in our own development work. In an elevator control simulation, we noted that the elevators never went to the top floor of the building; this turned out to be a boundary error on an array. In a Dining Philosophers program, each "philosopher" suddenly had two "forks" due to failure to remove a tuple. When this was corrected the program deadlocked with each philosopher holding one fork, due to the accidental use of the variable "j" where "i" was intended. Sometimes visual errors were due to incorrect formulation of the visualization rules. One instance occurred in our Bagger program, when the rectangles representing items initially failed to align with those representing bags; the visualization rule that computed the position had been written to sum the weights of all items inclusive of the last item, while the correct sum would be exclusive of the last.

We have also performed several experiments in which, having developed a correct set of visualization rules for a particular application, we deliberately perturb the underlying computation to cause an error. Our goal was to see if the visualization rules would provide us with an indication of the problem. One such experiment was performed with a Diffusing Computations visualization. The underlying computation deals with termination detection in a network of processors which communicate by message-passing. The algorithm makes use of a system of acknowledgments whereby each processor eventually sends one acknowledgment for each message it receives. The error we introduced was to have one processor send an acknowledgment before it should have. Figure VE-1 is the result. The spheres are all supposed to be linked into a tree structure; however, because of the error the tree structure has been broken, leading to immediate visual confirmation of the error.