Reshapeable Visualizations

Mihail E. Tudoreanu  
Washington University  
St. Louis, MO 63130  
renu@cs.wustl.edu  

Delbert Hart  
The University of Alabama  
Huntsville, AL 35899  
dhart@cs.uah.edu  

Gruia-Catalin Roman  
Washington University  
St. Louis, MO 63130  
roman@cs.wustl.edu

Abstract

Visualization is an effective tool for understanding the execution of complex programs. Over the course of a visualization session a user may employ a wide range of graphical representations. Adaptable visualization displays are needed to meet these diverse interests and applications. This paper presents an approach that supports interactive creation and refinement of visualizations via on-screen manipulation of existing visual representations of the program. By acting upon visualizations on the screen, users control what aspects of the computation and what properties are displayed. An action modifies the visualization in a given program state, but users need to have a means of defining how the visualization reacts to changes in the program state. We present a manner of specifying the effect of actions on a visualization for future program states. To achieve their effect and update the program representation, actions are automatically re-applied, in the order they were performed, for each state of the computation. The interpretation of user action is based on a model for reasoning about visualizations that focuses on what information is visually communicated and is independent of the exact choice of graphical elements. We define a set of operations on the model that allow users to take advantage on the information on the screen and explore related parts of the computation.

1. Introduction

Visualizations can be categorized on a continuum between being application-independent and application-specific. Application-independent visualizations are useful because they present the same information about any computation, thus reducing any learning curve associated with understanding the visualization. Application-specific visualizations are useful for seeing properties and behaviors particular to the computation being studied. The drawback of application-specific visualizations is that they need to be created specifically for that application. This overhead can prevent them from being used. Ideally, these visualizations could be created on-the-fly, with minimal effort.

We have developed a system to support the creation of application-specific visualizations and an underlying model upon which the system is based. Our approach is to allow new custom visualizations to be created by tweaking existing visualizations. This refinement process occurs through direct manipulation of graphical objects present in the visualizations. The development of the system required the definition of a semantic model to capture the effects of user actions. The model captures the information being conveyed by a visualization without considering how the information is communicated. The model is not interested in the specific graphical attributes used to create the picture, rather in how these attributes fit together to represent the computation. The main contribution of the model is the manner in which it maintains the user’s context as the computation executes. This is done by keeping a history of user’s operations to be re-applied upon each change in the computation. By defining how the operations are re-executed, users control the evolution of the visualization based on the evolution of the program. The model allows the behavior of an operation to be dependent on previous states of the computation.

This paper builds upon work done in both information and software visualization. Information visualization studies techniques of visually conveying various types of data. Specifically, this work builds upon the area of automatic data visualization ([4] [8] [2]) and employs the underlying concept of separating what is being communicated from how it is communicated. To create an animation, it suffices for users to specify the information they wish to present. The method of automatically generating the graphical representation is beyond the scope of this paper.

Software visualization presents how the execution of a computation unfolds. Users specify the graphical elements to be displayed for events or states of the program. Lens [5] is most similar to our work. It allows the creation of visualizations at run-time via direct manipulation. Users in-

\footnote{This work was funded by NSF grant CDA-9619831.}
teractively specify how the attributes of graphical objects are derived from the program state or other attributes. Animation annotations are introduced at different points in the program text. In our approach users work with the state of the program, not with the program code. This reduces the dependency on the control structures (e.g., constructs like \texttt{if}, \texttt{for}) of the program. During the refinement of the program presentation, we also allow users to take advantage of the underlying information associated with graphical objects.

A number of tools for creating application-specific visualization of running programs have been developed. Users write programs to produce graphical objects and define how attributes of those objects change with the computation. Zeus [1] is based on “interesting events” generated by the running application. These events are used to update a set of views (animated pictures). POLKA [10] employs the path-transition paradigm and can program graphical objects to perform actions at given animation times. A front-end interpreter, called SAMBA [9], can be used to translate text commands into animation actions. Pavane [6] implements the declarative approach; users construct a visualization as a mapping from the program state to an abstract animation. The mapping is then re-applied after each step of the program.

Direct manipulation of visualizations also appears in Visage [7]. Visage introduces the information-centric approach and allows the transfer of data between graphical representations. The purpose of Visage is to provide coordinated access to multiple presentation tools. This paper employs direct manipulation of visualizations as the mean of creating and refining program animations.

The next section presents the model. The system and examples of its use are presented in sections 3 and 4, respectively. A summary is given in section 5.

2. Visualization model

Online visualization provides a dynamic perspective on a running computation allowing users to better understand the behavior of the program. Given the size and complexity of many computations it is unlikely that visualizations will be available a priori to cover everything that might be of interest. This makes it desirable to allow custom visualizations to be created at run-time. In our approach, users interactively create and refine program visualizations through direct manipulation of graphical representations.

The nature of direct manipulation, i.e., the elements being changed and the possible set of modifications, depends on how graphical representations are thought of. Classical ways of reasoning about pictures, such as a matrix of pixels or as a set of graphical objects, expose users to incidental complexity of the visual representation. The extra information makes it more difficult to model what is being communicated in the picture and to modify it. It is desirable to distinguish which graphical properties are semantically meaningful from those that are purely aesthetic.

Our visualization model abstracts away from the choice of graphical attributes. The data and relationships presented by means of visual attributes are the elements available for users to manipulate in a picture. The novelty of our approach is that it regards the mapping of program states to pictures (a general definition of a program visualization) as a two step process. In the first step, the data and structure of the visualization are constructed. The result is an instance of our model. In the second step, graphical representations are automatically assigned to that instance. Users can modify either step, but they are required to define only the first. Interactions with pictures allow users to navigate between classes of visualizations, where elements of a class convey the same information using different visual features.

2.1. Scenes

A visualization can have multiple windows, each modeled by a scene. A scene describes the information communicated by a graphical representation, abstracting away from the particular graphical features used to portray the computation. At any particular time, the scene contains information derived from the current as well as previous states of the computation. A scene consists of scene elements, each encapsulates a piece of information that appears as one entity in the context of a graphical representation (e.g., a simple or composite graphical object). A scene element consists of a set of data specifications and a set of structural specifications. Structural specifications capture a scene element’s relations with other scene elements. These relations are expressed through the similarities and differences of the visual attributes (e.g., color, shape, position) corresponding to each element. Data specifications describe the subset of the program state depicted by this element.

Structural specifications capture the interrelationships between different parts of the visualization. The structural specifications of a scene are described as named relations of the form \texttt{[name = value]}. There exists a relation among scene elements that contain a structural specification with the same name. A logical equivalence is assumed among elements with the same [name = value] pair, while a logical distinction is being made between elements with the same name and different values. Any existing ordering on the domain of the values is extended on the elements. In the upper row of Figure 1, processes are grouped in two classes by shading. The structural specifications describing this relation is [marked = true] for A and [marked = false] for B and C. A scene element can be involved simultaneously in multiple relations.
Graphically, a relation between scene elements is shown through at least one graphical attribute. Within a relation, the objects corresponding to scene elements that are logically equivalent have identical values for the graphical attribute(s) of the relation. As such, the graphical objects have some visual characteristics in common. Graphical objects for logically distinct scene elements have different values for the attribute(s) depicting that relation.

The state is regarded as a set of tuples (tuple space). This space continuously changes through atomic steps as the computation runs. *Data specifications* describe what part of the state of the computation is represented by the scene element. Data specifications help users identify the scene elements depicting a certain kind of data, which is practical for presenting that data in more detail or exploring related parts of the program. Note that structural specifications do not directly express the subset of the program state that is being visualized because they are derived values.

### 2.2. Temporal behavior of operations

A scene describes a picture that corresponds to one state of the computation. Users construct a scene by starting with a visualization and modifying it through a sequence of operations. While the computation is running, the scenes of the visualization are automatically updated to reflect the changes in the application state. Users determine how the visualization responds to changes in the program state by specifying the behavior of individual operations over time. The operations are kept in a history list that defines the mapping of the underlying computation to scenes. When the state changes, the history is re-applied from the empty visualization.

For one state of the computation, the user’s operations on a scene can be given precise semantics in terms of what scene elements and specifications are modified and what their new values are. These values are specified as a function of the program state(s) and of the current content of the scene. In the context of an evolving computation, an operation can be re-applied in multiple ways based on what set of states the operation will depend on in the future. There are three modes of re-applying an operation: absolute, relative and cumulative. The mode is chosen by the user at the time the operation is first executed. For simplicity, only operations that perform selection, i.e., choose operands for future processing, are allowed to have any mode, all other operations work in relative mode.

For example, suppose that the user wants to highlight idle nodes in a computation that has three nodes, and they become idle in the order A, B, C with only one idle at a time. The first time the operation is executed, A is marked (see Figure 1, first column). After the program state has changed, one possible interpretation of the operation might be to display the same process that was initially idle, A, even if it is not idle anymore (first row of the figure). Another possibility would be to ignore the past and show the process that is idle now, node B (second row). There is yet another option: to present both processes that were idle and the ones that are idle. More precisely, processes that were idle in any state between the initial and the current one.

A selection operation in absolute mode has the same effect regardless of the current state. The same items are selected as when the operation was first applied. Such an operation is not directly recorded in the history. An operation that specifies exactly the instances of selected items is recorded instead.

A relative operation does not depend on the previous program states, only on the current one. Such an operation behaves as if the user executes it for the first time every time the state is modified. It is added to the history in its original form. Non-selection operations, the ones that modify scenes, are always in the relative mode.

Finally, a cumulative operation depends on the sequence of states that has occurred since its first execution. It is similar to having that operation performed in absolute mode for every state in the sequence. Two operations are added to the history. One is the operation as if in absolute mode and the other is the operation itself, in cumulative mode. When the history is re-applied, the second operation causes another operation in absolute mode to be recorded between the first and the second. The history will contain one absolute operation for each state since the operation was performed.

New operations can be defined by combining a sequence of operations into a single, atomic one. A sequence that contains selection in cumulative mode is recorded once for each state of the computation, with the selec-
tion operation(s) in absolute mode for the corresponding state. This ensures that the atomicity of the operation is preserved. In the example, first idle nodes are selected (operation \( Sel(idle) \)), a new structural specification is added to them (operation \( Mark \)). The sequence would be recorded as \( < Sel(A), Mark > \) in absolute mode, as \( < Sel(idle), Mark > \) in relative mode, and as \( < Sel(A), Mark > < Sel(B), Mark > < Sel(C), Mark > \) in cumulative mode.

3. The Reshapeable Visualization System

The Reshapeable Visualization system implements the framework described in the previous section. The system provides an environment in which users can build graphical representations of a computation through direct manipulation of scenes. The information and structure of a visualization is defined by user actions on scenes. Pictures corresponding to those scenes are automatically drawn by the system. Users can intervene and determine entirely or partially the graphical elements to be used. Due to space limitations, only an overview of the architecture and of the user’s means of controlling it are presented. Other challenges of the system such as rendering heuristics, editing and pruning the history of operations or defining graphical objects are beyond the scope of this paper.

The scene generator processes user operations and records them into a history as defined in the previous section. The state of the computation is provided by the monitoring system PathFinder [3]. PathFinder notifies the scene generator when the computation changes and provides the data describing the state of the computation. Upon notification, the scene generator re-executes the history.

The rendering system chooses a graphical object for each scene element in a scene. The system also assigns graphical attributes and values to the rendering specification such that the final picture presents the structure of the scene. Users can specify the exact graphical elements to be assigned to a scene element or structural specification, or they can override the decisions of the rendering system. In the former case, the names and values that appear in the structural specifications can influence the system decisions. The system interprets a structural specification of the form \( \text{SHAPE} = \langle \text{ObjName} \rangle \) as a user request to select the graphical object named \( \text{ObjName} \) to depict that scene element. The system also tries to match the name or value of a structural specification with the name of an attribute. As such, a specification \( \text{width} = 3 \) is assigned to the width attribute if possible (there exists an unassigned attribute named \( \text{width} \)).

The rendering system uses Java3D to draw the graphical objects on a 3D canvas. Navigation capabilities, zooming, translation and rotation of the point of view in the 3D space, are available to users. The transition from a picture (state) to another is performed smoothly by interpolating linearly between old and new values of attributes.

4. Manipulating visualizations

Visualizations are manipulated through three types of operations: selection, refinement and meta-operations. Selection operations allow users to choose items, i.e., scene elements, structural specifications or tuples, that will be used in subsequent operations. Items of interest are selected by specifying each instance or a common property of the items. Refinement operations modify the scenes of a visualization by changing their components, deleting or adding new components. Meta-operations allow users to extend the available set of operations based on already existing ones. A sequence of operations can be aggregated into a composite one or can be applied repeatedly as part of an iteration.

The use of operations is illustrated via a sample visualization session. A hypothetical user investigates how a railroad system functions. The railroad components (trains, signals, and switches) are ‘smart’ devices that communicate with each other to assure the safe passage of trains to their destinations. The railway is divided into segments, each guarded by a signal. To enter a segment, a train sends a request to the corresponding signal. If the request is approved,
the train moves onto that segment. Otherwise, it either waits or tries a different route. The computation to be monitored is a simulation of this railroad system.

The types of tuples in the program state and their corresponding fields can be observed in Figure 3. A track tuple represents a piece of track starting at position \((x, y)\) with the given orientation and status. Orientation expresses a compass direction like North, North-West, West, South-West and so on.

4.1. Visualization session

The user starts exploring the computation in a window named State View. The State View presents textually form the tuples of the current state of the program. The user has the option of monitoring all individual tuples or only the available types of tuples as in Figure 3.

![Figure 3. The types of tuples in the program.](image)

The user begins by creating a picture of the tracks, lights and trains of the computation. To obtain the graphical design presented in Figure 4b the user communicates the specific graphical elements (e.g., location, orientation) to the system via the names and values of structural specifications. The user also chooses what graphical objects are used.

![Figure 4. On the left, a snapshot of the trains moving on the railroad. On the right, the dialog for creating the track scene elements.](image)

To display the tracks, the user chooses all tuples of type track then clicks on the corresponding type in the State View. Selection operations can choose scene elements, structural specifications and tuples. For each of the three types, there exists one set containing the selected items of that type. The three represent distinct concepts that are used as input for operations.

Once the data of interest is chosen, it can be graphically shown. Selected tuples can be included in a scene element and depicted on the screen via a refinement operation named create element. In the Create Element Dialog (Figure 3), the user can choose whether each tuple appears as a separate element or all tuples as one element. The dialog can be used to determine the structural specifications of each new element. The value of the specification is either constant or dependent on the tuple(s) of each new element. The value of a tuple field is referenced using the special symbol ‘$’ that prefixes the field name. In Figure 3, value Line remains the same for all new elements, while orientation is replaced by the actual value of orientation field of the track tuple. As a result, each scene element depicting a track is a line and has a structural specification describing its orientation.

After the dialog is executed, the tracks forming the railroad appear in the visualization. The user adds the signals and trains to the visualization in a similar manner, via selection and scene element creation. To ensure that the status of signals and position of trains change with the computation, the operations are executed in relative mode.

The visualization described so far does not offer information about the route followed by a train. Now we will consider how to modify the visualization to show such information. For each train, it is enough to add a structural specification \([\text{visited} = x]\) to a track that has been visited by that train, where \(x\) is the number of the train. The same structural specification needs to be added to the train object so the route and the train share a graphical characteristic. No specific graphical elements need to be given by the user, they are automatically assigned.

It is cumbersome to perform the same task for each train, especially if new trains appear. So, the user shows how to mark the trail for a train, and the system iteratively applies it for all trains. Iteration requires as input a range and a sequence of operations, and it re-executes the sequence for every element of the range. The range is considered to be one of the selected sets, i.e., scene elements, structural specifications or tuples. The sequence of operations is specified by means of an example, applying a sequence of operations to one of the items in the range.

The iteration sequence of operations is presented below.

1. A train object, part of the iteration range, is selected.
2. The tuple of type train contained in the data specifications of the train object (step 1) is selected.
3. The track object at the trains current location is selected in cumulative mode.

4. A structural specification \([\text{visited} = \text{nm}]\), where \(\text{nm}\) is defined as the value of the \(\text{number}\) field of the selected tuple (step 2), is added to the structural specifications of the selected objects (train and track).

Notice that because the track is selected in cumulative mode, a new iteration operation is added to the history for each state of the program. Each such operation marks the tracks that had a train on them in the corresponding state.

Instead of manually selecting train scene elements, the user can specify the property of the elements of interest. Property-based selection allows users to inspect aspects of the computation related to the current visualization. Textual input can be introduced in a Selection Dialog, or items of interest can be dragged to the dialog. The property given by the user is \(\text{tuple.type=\text{train}}\), which was obtained by dragging the \(\text{type}\) field of a \(\text{train}\) tuple. The user activated the selection of scene elements. This instructs the system to select all scene elements that contain a tuple whose \(\text{type}\) field has the value \(\text{train}\).

The user adds the structural specification \([\text{visited} = \langle \text{nm} >]\), where \(\langle \text{nm} >\) is the train number, to the currently selected scene elements. Change Structure Dialog provides the functionality of adding to, removing from or replacing the structural specifications of an element. The train number is determined via reference to the \(\text{number}\) field of the select tuple, i.e., \(\text{visited} = \$\text{tuple.number}\). In this scenario, only one field with the given name was present in the selected set. If more than one field existed, multiple structural specifications would be added, one for each field.

The visualization, after the iteration, presents all trains moving on the railroad. As they move, their trail is marked by a different color, each corresponding to a different train. Notice that the trails do not appear in the state of the computation, but they are computed by the visualization system.

In general, the Selection Dialog is used to select scene elements. For selection purposes, data specifications of an element are viewed as a collection of name–value pairs that correspond all fields of the tuples. Hence, both the structural and data specifications have the same format: a name–value pair. Scene elements of interest are specified with constraints of the form \(\langle \text{name} > [\langle \text{op} > \langle \text{value} >]\) where \(\langle \text{name} >\) is the the name of a pair, \(\langle \text{op} >\) can be equality, difference or a comparison, and \(\langle \text{value} >\) is an expression. Each element in a scene is checked to see if it satisfies the constraints. The search for a matching name can be restricted to only tuple fields by specifying the prefix ‘tuple.’ in \(\langle \text{name} >\), and to structural specifications with the prefix ‘struc.’. A ‘$’ in \(\langle \text{value} >\) can be used to reference an already selected item.

The Selection Dialog can be also used to select tuples from either the computation state or data specifications of selected scene elements. Similarly, structural specifications can be selected from the already selected scene elements.

5. Conclusion

Supporting the creation of and interaction with application-specific visualizations through direct manipulation enables the use of visualizations tools for many who otherwise may not have the technical expertise to create custom mappings from a program execution to animations. The Reshapeable Visualization system accomplishes this task by using a history list of operations to maintain the user’s current view of the computation while it runs. The system is based on a clean conceptual model that separates what information is to be communicated from how it is shown.

References


