Paramour: Managing Complexity in a Visual Data-Flow Environment With Enclosures and Busses

Simon Greenwold
Massachusetts Institute of Technology, Media Lab
simong@media.mit.edu

Abstract

This paper introduces Paramour, a visual data-flow programming system for the development and deployment of interactive graphic scenes. The system implements two features designed to increase the visual clarity and manageability of programs written in it. "Enclosures" allow subgraphs of arbitrary size to be abstracted, stored, and reused. Enclosures can express as inputs or outputs any number of parameters of any supported type. "Busses" solve the problem of visual clutter when many entities desire access to the same few values by effectively creating a visual namespace to carry such values across the top of an enclosure, from which they may be pulled down for use in computation.

1 Introduction

There is no good name for an interactive graphic scene. An animation is a two- or three-dimensional image parameterized in time, but when images are parameterized in many dimensions, often nonlinearly in response to user manipulation, we have no suitable term to describe the product. Yet such interactive scenes are increasingly found as rich user-interfaces, games, or customized media. The ultimate potential of the interactive scene is to become software in itself, blurring the line between program and interface.

Tools for the creation of such active content exist (Macromedia Flash and Director, for instance), but any flexibility they offer in the behavior of entities comes by way of textual, imperative scripting languages. This requires developers of interactive content to learn procedural programming in order to exert control over behavioral aspects of their work. One of the few exceptions, a program called Geometer's Sketchpad [1], offers easy construction of complicated relational geometry, but offers no tools for visualizing or editing the implicit data-flow graph.

As others have recognized [2], a visual data-flow language is a perfect match for this domain because the entities that construct such interactive scenes—points, lines, polygons, text, and images—are well defined, visible, and easily parameterized. Further, the parameters of these entities fall neatly into a few primitive data types: Boolean, number, string, location, and color. Given the neatness of the domain and the obvious benefit of parametric relationships to specify design constraints (e.g. an arrowhead should remain on the end of a segment), Paramour offers real benefits for the construction and maintenance of complex interactive scenes. We have used Paramour to implement and publish to the Web several teaching modules for elementary structural analysis.

2 Paramour system

Design of scenes in Paramour takes place on two canvasses: the scene editor and the scene graph editor.

2.1 The Two Editors

The scene editor is similar to traditional graphic design applications: tools for drawing points, lines, buttons, and text are available. Every graphic entity created also appears as a rectangular node in the scene graph editor, in which the relationships between parameters, functions, and enclosures can be manipulated.

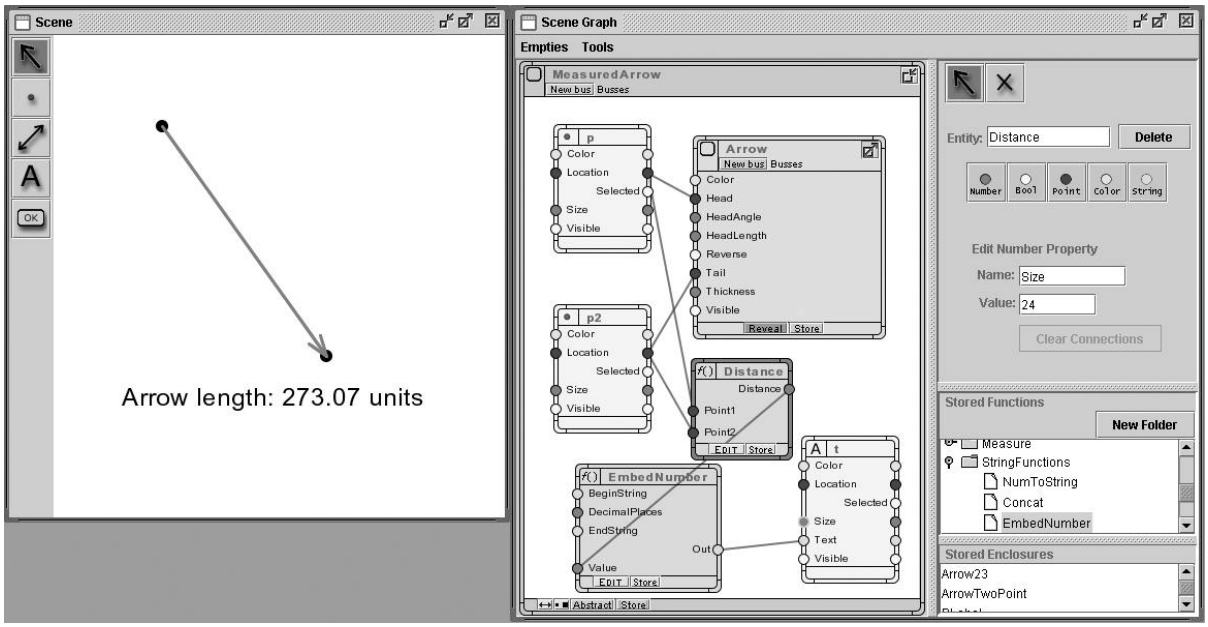

Figure 1. A simple Paramour session. On the left is the scene—an arrow which can be manipulated and text that dynamically indicates its length. On the right, the scene graph that determines the scene.

There are three classes of entities represented in a scene graph: enclosures, graphic entities, and functions. All entities appear as resizable rectangles titled with their names. Properties of entities are displayed as small circles colored by property type and named. Input properties are arrayed along the left side of entities, and outputs to the right. To bind an input property to the value of an output property, a user simply drags a line from one property node to the other. A blue line indicates that a connection has been established. Graphic entities and functions are permitted to contain properties called “pass-throughs” which are both inputs and outputs. Pass-throughs always output the same values that they receive as input (except in the case of “constraints” discussed below).

2.2 Graphic Entities

Graphic entities are created in the scene editor and correspond to any visible element of the scene. All graphic entities can, however, be hidden or shown by means of a Boolean visibility input property they contain. A user is free to select and move graphic entities in the scene editor. Any graphic entity with a non-bound location input property or a location input property bound only by a series of pass-through properties is susceptible to user manipulation. The dragging of a graphic entity in the scene changes the location property’s value. If a location property is bound by a series of pass-through properties, the unbound (first) pass-through is set when the graphic entity is dragged. This allows some degree of back-propagation of changed values when the backwards path is known to require no inverse functions.

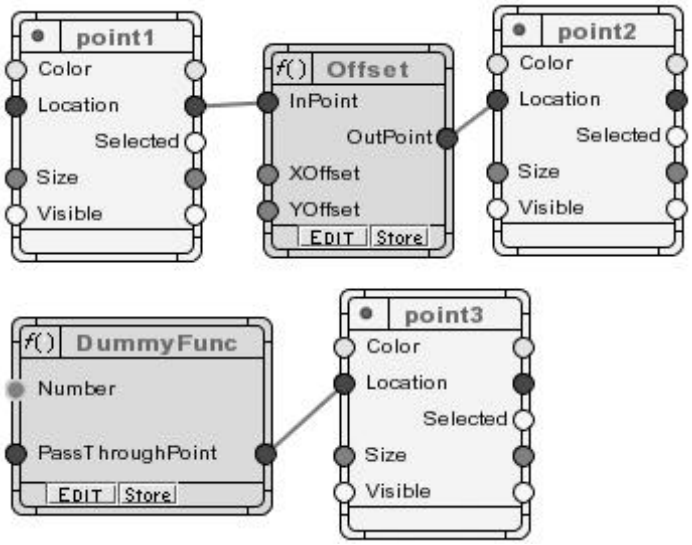

Figure 2. "point1" can be dragged freely because its "Location" input is unbound. "point2" will move if "point1" is dragged but cannot be dragged itself because its location is bound to an output of the "Offset" function. "point3" can be dragged even though its location is bound to an output of "DummyFunc" because the property it is bound to is a pass-through.

It is often the manual rearrangement of graphic entities that drives the action of a scene. For instance, a line of text may read the length of a line segment between two moveable points. All graphic entities, and indeed all entities are contained inside an "enclosure." The scene graph highlights the currently active enclosure, into which all new graphic entities are automatically placed.

2.3 Enclosures

Enclosures are a mechanism for encapsulation, abstraction, storage, and reification of subgraphs. Every entity created belongs to an enclosure. Enclosures themselves belong to other enclosures, creating a hierarchy which originates from a root enclosure—the scene in toto. Entities with the same direct enclosure must be named uniquely. Enclosures may contain any combination of entities connected in any fashion. Connections that are drawn between properties can occur only at the same level of enclosure; that is connections may never cross enclosure boundaries and therefore introduce a notion of data hiding. The interior organization of any enclosure is inaccessible to the outside and can be guaranteed to remain valid regardless of its external context. This environmental invariance is a key condition for useful reification of stored enclosures.

Values are passed into and out of enclosures by means of properties expressed on their boundaries. A user may choose to make an enclosure express any number of properties as inputs or outputs. To do so, he drags a property of the appropriate type from a palette onto the left (input) or right (output) side of an enclosure. He then names it and connects it to a property of an entity contained in the enclosure. This way an enclosure can be seen to represent a potentially multi-valued function that may define graphics. Enclosures can be "abstracted," meaning that their inner graphs are obscured, making them appear to be simple atomic entities. Conversely they can be revealed and even maximized to take up the entire space of the scene graph so that all subsequent editing takes place inside the currently active enclosure.

This hierarchical complexity hiding requires a user frequently to “drill down” into a series of nested abstracted enclosures to access certain entities or properties.

The context-independence of enclosures makes them an ideal vector for storage and reification of subgraphs. Any enclosure may be named and stored. It will appear in a list of stored enclosures and can be dragged out of the list to be instantiated in any number of copies in any context. Since the uniqueness of entity names is required only at the same level of enclosure, the only name that ever requires adjustment is the name of the outside stored enclosure, which sometimes requires an automatically generated suffix to be applied to make it unique in its new context. If a stored enclosure contains graphic entities, reifying the enclosure creates new graphic entities with the same property values and connections, offering a convenient way to define and encapsulate parameterized geometric constructs such as smart arrows or custom controls.

2.3 Functions

Functions, the final class of entities take advantage of the system’s ability to dynamically compile and link Java classes. Functions taking any combination of inputs and producing any variety of outputs can be defined by dragging properties to the function’s graph node exactly the same way properties are added to an enclosure. The properties are then named uniquely and the function code can be edited in place in the graph. Because we are able to place the function code in context before compiling it, the function can refer to its inputs and outputs by name using any valid Java syntax. Functions can also be stored, retrieved, and modified. Time-based animation is accomplished by setting up Java Timer objects with callbacks into the function’s code that updates the outputs. This way generic animation functions can be defined, such as “MoveInACircle,” which can be parameterized and attached to any location property in the graph.

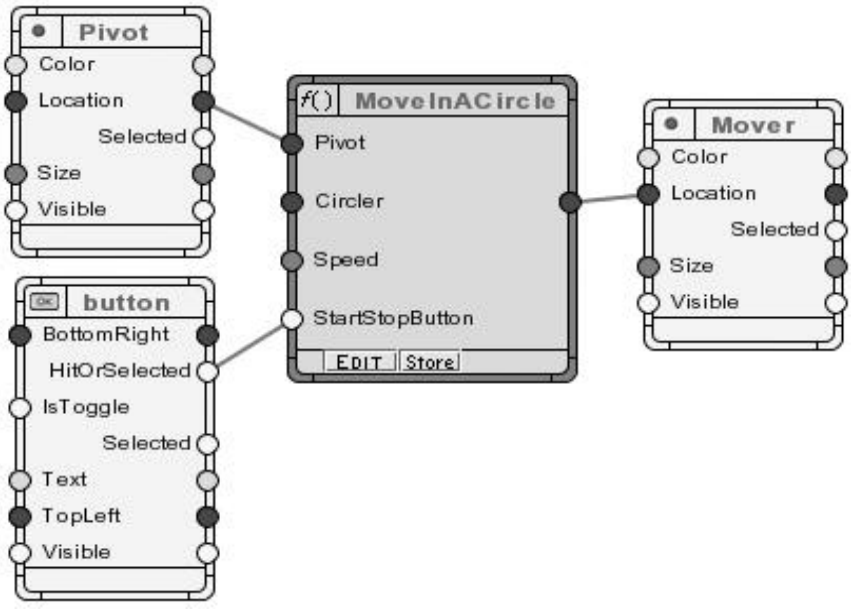

Figure 3. This is a graph for a scene that makes one point circle another when a button is pressed.

2.4 Busses

A problem with Paramour's graphs' clarity became evident in early use as we developed some simple structural analysis demos. Certain values such as the physical geometry of the system were necessary for almost every enclosure to be aware of. Since the encapsulation of the enclosures was so strict, disseminating these commonly used values meant expressing them as outputs on every enclosure above where their graphic entities were defined and then connecting them up the hierarchy ladder. Then all other enclosures that needed them had to subscribe to them from the same few property nodes. This created a visual snarl in which a single property node had twenty or more connections emanating from it, making it impossible to easily understand what was going on. Some have proposed topological organization methods to deal with this problem [3], but we have examined a method that circumvents the situation entirely.

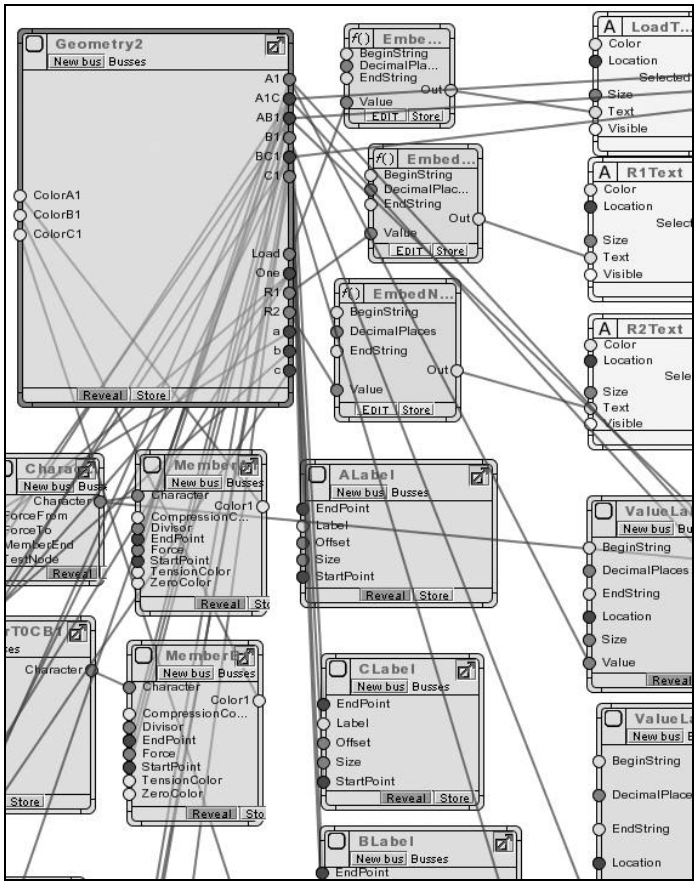

Figure 4: A part of the scene graph for a structural analysis demo shows the difficulty of disseminating commonly used values. Virtually all enclosures need to have access to the basic geometry of the figure that is being analyzed resulting in a web of connections that is difficult to work with.

Any enclosure may define and name one or more "busses." These appear as bars across the top of the enclosure. Any output property directly enclosed can be dragged up to a bus, and a "wire" will appear in the bus carrying that output's value. Connections can be made to input properties in this enclosure from anywhere along this wire.

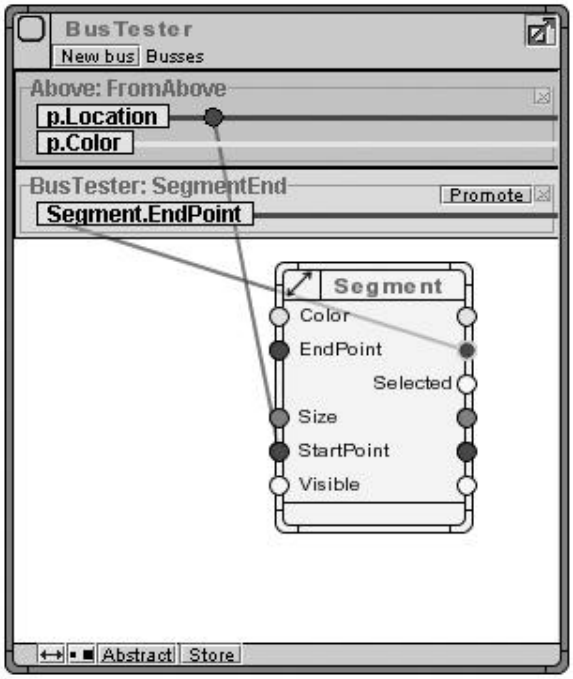

Figure 5: This enclosure "BusTester" subscribes to one bus called "FromAbove" defined in an enclosure above it. It draws the start point of a line segment from this bus. It also creates a new bus called "SegmentEnd," which carries a wire with the value of the end point of the segment.

Busses are automatically accessible to all enclosures descended from the enclosure in which they are defined. They cannot be modified or added to by any other enclosure, but this way values can be offered to children of an enclosure without explicitly passing them. This scoping mechanism resembles C-style block scoping rules, and comes with the same issues. For instance, a stored enclosure that subscribes to a bus defined above it might not be reified into a context that defines such a bus. In an instance such as this, a user is warned, and any property connections to a missing bus are simply not restored.

Downward scoping eliminates half of the problem of commonly used values, but it is also sometimes desirable to allow public access to values created at a very low level in the hierarchy. In order to accomplish this, any busses that are made visible in an enclosure can be "promoted," making them accessible to the enclosure directly above the promoting enclosure. This also makes the bus accessible to all enclosures below that one, effectively allowing a value to be disseminated to any enclosure that is a sibling of or descendant of a sibling of the promoting enclosure. A bus can be promoted more than once, essentially percolating values up to a level in the hierarchy from which they can be used by all enclosures that require them.

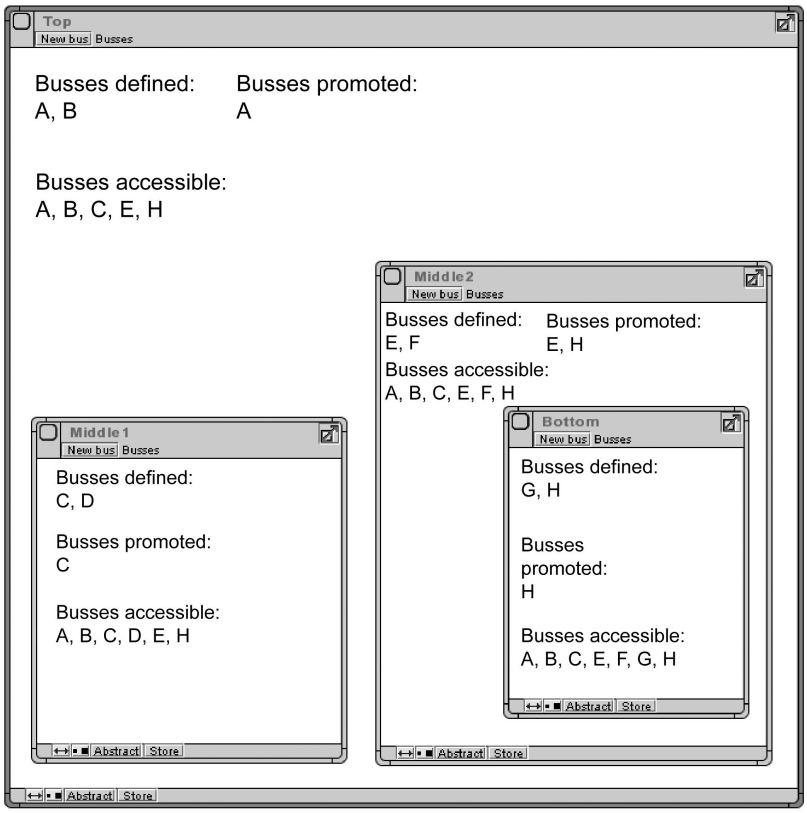

Figure 6: This is a conceptual diagram showing the rules of bus accessibility and promotion in nested enclosures.

Busses have shown to be an effective way to reduce visual clutter while violating beneficial encapsulation as little as possible.

2.5 Constraints

A problem of partial constraints arises in a system such as Paramour, which seeks to allow interactive control of a dependency graph. Some properties must be changeable but only in certain ways. For instance a user commonly wants a point constrained to lie somewhere on a line segment. A user should be free to pull the point along the segment but not off it. If we bind the value of the point's location to some calculation of the graph we lose all of its freedom to move, and if we leave it unbound, we cannot constrain it at all.

The solution we implement in Paramour is due to the mechanism of back-propagation that occurs when a location property is bound only by pass-through properties. We allow functions to define a special section of code called constraint code, which modifies pass-through values when they are retrieved. That means that although a property is a pass-through, it may show a different value upon retrieval than the value that was passed to set it.

For example, when a point's location is bound to a function that constrains locations to their nearest points on a segment, the location property of the function is a constrained pass-through. Any dragging of the point will set the value of the pass-through property, but when the value is retrieved by the point, it will have been filtered through the constraint code and will stick to the segment.

2.6 Implementation

Paramour is implemented entirely in Java, which has several advantages. First it was quick to develop, and much of the interface and drawing work is done using Sun's SDK. The most important advantage, however, is in the deployment of interactive scenes created with Paramour. Any stored enclosure can be selected and published to a Java applet, which can be embedded in a web page with all of its behaviors intact.

3 Future work

Thus far Paramour has proven useful for the creation and deployment of simple interactive scenes. Certain improvements in interface could significantly aid workflow. For instance, automatically opening the scene graph to the enclosure of the selected object in the scene would save a great deal of time spent drilling down to uncover an entity.

3.1 Compound Types

The most important conceptual addition would be support for custom compound data types. With compound types much of the redundant effort involved in passing related values could be avoided. For instance, every function that takes a line as an input requires two separate point values to be passed. It would be helpful to be able to define a data type consisting of two points and simply pass that. Further, it would be interesting to allow types to be full classes by letting them contain methods that are run to produce outputs.

It is possible that a good way to introduce compound types would be the passing of entire entities to properties. A property would specify what output conditions an entity passed to it must define, and any entity that does so would qualify. Passed entities could include functions, graphic entities, or enclosures.

3.2 Self-Modification

Right now Paramour is a tool for describing steady states. All the action of a scene is involved in keeping it consistent with a single graph. There is no facility for the graph to describe changes to itself. Imagine that a certain application calls for the creation of an arbitrary number of points determined by a user's interaction with the scene. How can a graph describe the creation and integration of new nodes into itself? We must do further research to determine if there is a good way to accomplish this visually, perhaps through state-transition diagrams [4], or if we must fall back on textual languages to describe changes to the graph as in [5].

4 Conclusions

Paramour goes a long way toward making a useful system for the creation and maintenance of interactive scenes. It implements enclosures as a hierarchical organizing principle to limit the visual complexity of graphs and to introduce mandatory encapsulation. Busses soften this rigid encapsulation by allowing values to be grouped into named constructs and passed up and down the hierarchy with little graphical clutter.

The ease with which Paramour creates and disseminates simple interactive scenes gives us great hope that visual languages will soon be widespread in the domain. Graphic designers will be able to mold complex behavior as easily as they do form.

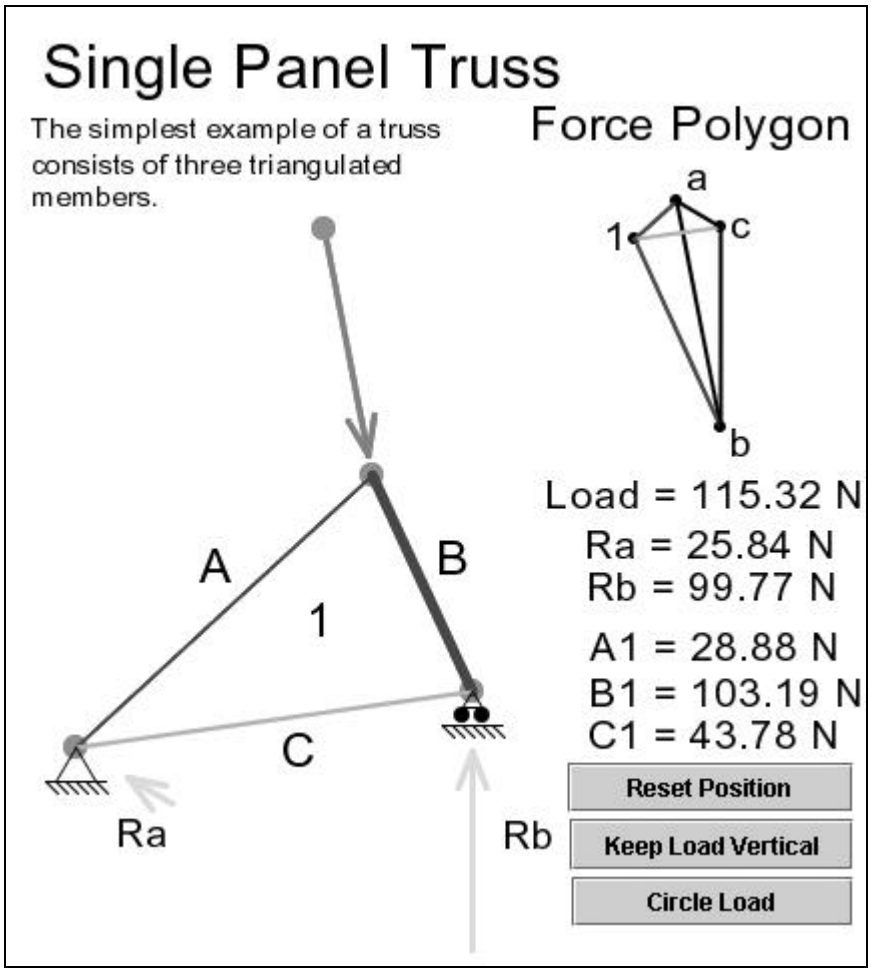

Figure 7: A simple structural analysis scene made with Paramour. This demo can be run as an applet from *acg.media.mit.edu/~simong/panelDemo/*

References

1. M. Battista. *Shape Makers: Developing Geometric Reasoning with The Geometer's Sketchpad*, (Berkeley, CA: Key Curriculum Press, 1997).
2. C. Elliott, G. Schechter, R. Young, and S. Abi-Ezzi. *TBAG: A high level framework for interactive, animated 3D graphics application*. In Proceedings of the ACM SIGGRAPH conference, 1994.
3. B. Ibrahim, Y. Hidenori. *Solving the Spaghetti Plate Syndrome in a Control-Flow Language with a VLSI-Like Solution*. In 1999 IEEE Symposium on Visual Languages (VL'99), Tokyo, Japan, September 1999.
4. Robert J.K. Jacob. "*A State Transition Diagram Language for Visual Programming*". IEEE Computer 18, 8 (Aug. 1985), 51-59.
5. M.F. Kleyn and J.C. Browne. *A high level language for specifying graph based languages and their programming environments*. In Proceedings, 15th International Conference on Software Engineering, 324-334, May 1993.