ConMan: A Visual Programming Language for Interactive Graphics

Paul E. Haeberli

Silicon Graphics, Inc.
Mountain View, CA 94043

ABSTRACT

Traditionally, interactive applications have been difficult to build, modify and extend. These integrated applications provide bounded functionality, have a single thread of control and a fixed user interface that must anticipate everything the user will need.

Current workstations allow several processes to share the screen. With proper communication between processes, it is possible to escape previous models for application development and evolution.

ConMan is a high-level visual language we use on an IRIS workstation that lets users dynamically build and modify graphics applications. To do this, a system designer disintegrates complex applications into modular components. By interactively connecting simple components, the user constructs a complete graphics application that matches the needs of a task. A connection manager controls the flow of data between individual components. As a result, we replace the usual user-machine dialog with a dynamic live performance that is orchestrated by the user.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques - User interfaces, D.3.2 [**Programming Languages**]: Language Classifications - Data-flow languages, Nonprocedural languages; I.3.6 [**Computer Graphics**]: Methodology and Techniques - Interaction techniques, Languages;

Additional Key Words and Phrases: Visual Programming Languages.

Introduction

Often we think of a user interface toolkit as a set of facilities that a *developer* can use to shape the feel of an application. For example, to make a choice available a developer can use a pop-up menu or a screen button. But after the developer compiles an application, the user is left with a static user-interface that reflects the developer's vision. If the user's task doesn't fit into the developer's model, then the user must use a different approach or try to find another application that does a better job.

An alternative is to present *users* with a toolkit and let them match it to a given task. In the UNIX* world, there are lots of simple tools a user can combine to solve different problems. The mechanism that joins these tools is a pipe, a simple one-directional interprocess communication (IPC) facility. This is an approach where the power of the sum is much greater than the power of the individual parts. ConMan (Connection Manager) provides a conceptually similar graphical facility for connecting visually-oriented tools. With ConMan, developers can concentrate on the purity of simple components. With good components that perform individual tasks well, a user can find a combination to solve problems that the designers didn't envision.

To escape the mechanical world of tools and toolkits, we'll use the culinary metaphor of a sandwich. Conventional systems present you with a ready made sandwich. You can add mustard and relish, but most choices have been made by the sandwich maker and your job is to find a sandwich that is closest to your needs. ConMan gives you the ingredients for the sandwich and leaves it to you to design a good one. This glosses over an important point: if you aren't a good cook, then the sandwich won't be very tasty. This isn't entirely facetious - the tradeoff between an expressive system and a ready-made system will always benefit some users and leave others unsatisfied.

Background

Although there have been amazing advances in graphics display hardware in the last ten years, applications have been slow in using the new capabilities pro-

* UNIX is a trademark of Bell Laboratories.

vided by the current generation of interactive graphics workstations. The structure of interactive applications has changed very little.

A typical application is integrated and self-contained with a single process and address space. The user interface is compiled into the program, or read in from an external description as in [Schulert 85]. The behavior of the application is described by a textual language that is compiled into an executable program. Functional binding happens at compile time and is static.

Users are prevented from expanding the design space interactively because the scope of an application is often limited by the vision of its designer. Also, traditional graphics applications are anti-social because they don't play nicely with other applications.

These characteristics often result in the user being dominated by applications. Instead of the user driving an application, the user is often driven and constrained by the application.

We want to use the facilities of the modern interactive medium more effectively to give the user more expressive power and freedom to construct and modify applications in a flexible way. Why isn't application development more like making a bacon, lettuce, and tomato, cucumber, salami, avocado, Jell-O®† [Heckbert 87] and sushi sandwich? Can't we use the interactive medium itself to help us?

Visual Programming

Visual programming describes any system that lets the user specify a program using a two dimensional notation. Instead of editing a one dimensional stream of characters, the user interacts with a two dimensional representation. A good discussion of various visual programming languages is given in [Myers 86].

Smith's Alternate Reality Kit [Smith 86] is a dynamic simulation environment with a visual interface. Objects have mass, velocity and a visual representation. The user can interact with the objects and change how one object influences another.

Other interesting visual programming systems are described in [Kimura 86a], [Kimura 86b], [Cardelli 86], [Blythe 86], and [Galloway 87]. These use two dimensional data-flow constructs to describe program behavior. Kimura's system, *Show and Tell*, runs on the Macintosh computer. It's a general purpose system that handles pictorial and textual data. It has some interesting graphical constructs for conditionals and iteration.

Cardelli has developed a conceptual framework for a system he calls *Fragments of Behavior*. In his system, each fragment has an interface for communicating with other fragments and possibly a dialog for communicating with users. The behavior of each fragment is described in the *Squeak* language [Cardelli 85], which resembles Hoare's language for communicating sequential processes [Hoare 78].

† Jell-O® is a trademark of General Foods.

The systems by Blythe and Galloway use data-flow constructs to control music synthesis and design digital filters interactively.

Tanner's Switchboard [Tanner 86] supports flexible communication between a population of processes running under the Harmony operating system.

The World of ConMan

In ConMan, we also use a data flow metaphor. The user constructs and modifies applications by creating components that are interconnected on the screen. The window manager supports creation and deletion of individual components, while the user changes the interconnection by interacting with ConMan, the connection manager.

Figure 1 shows how this interconnection can be described by a directed graph with components as nodes, and connections as edges. Connections establish dependencies between one component and another. Each component can have up to eight input ports and up to eight output ports. By interacting with the connection manager, the user may alter this dependency graph at any time, without the knowledge of the components.

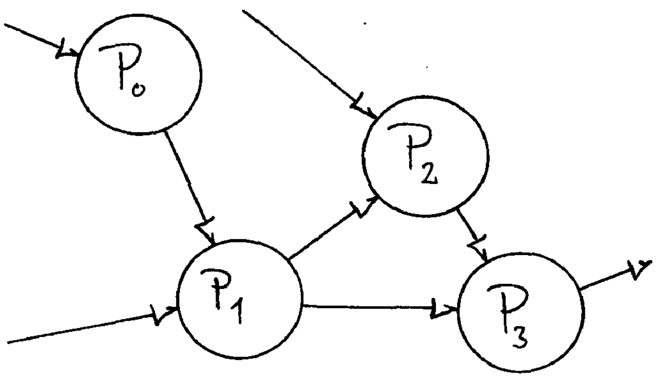


Figure 1. A directed graph representation.

Any dynamic interaction is easier to demonstrate than to describe. To show how ConMan works, we'll discuss a composite application that lets the user interactively design swept surfaces. This example will use six simple components:

- *view-ed with sliders*. This component controls the view of a surface with a set of sliders.
- *view-ed with hemispherical control*. This component allows the user to control the view of a surface with hemispherical control.
- *curv-ed*. A simple curve editor lets the user interactively enter or modify two dimensional shapes.
- *sweep*. The sweep component takes a shape, for example a curve from curv-ed, and sweeps it through space to create a surface.

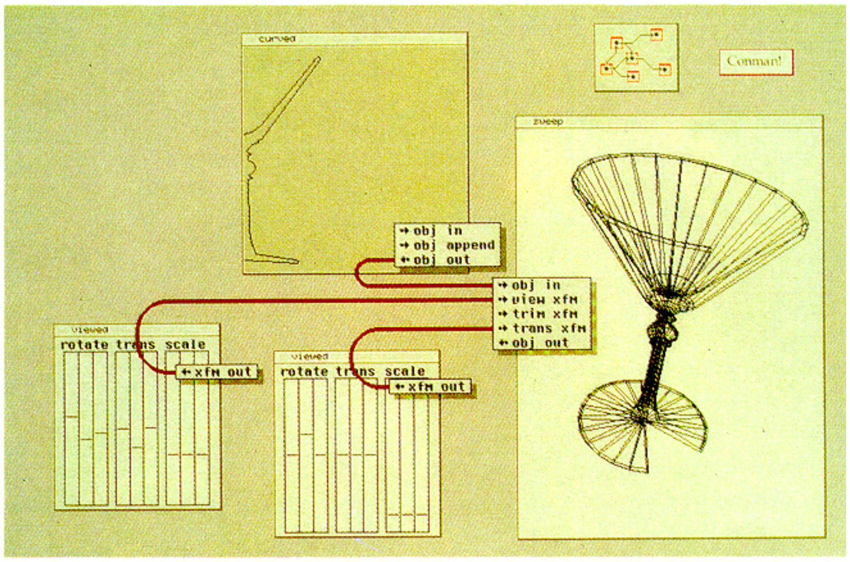


Figure 2. A simple ConMan application.

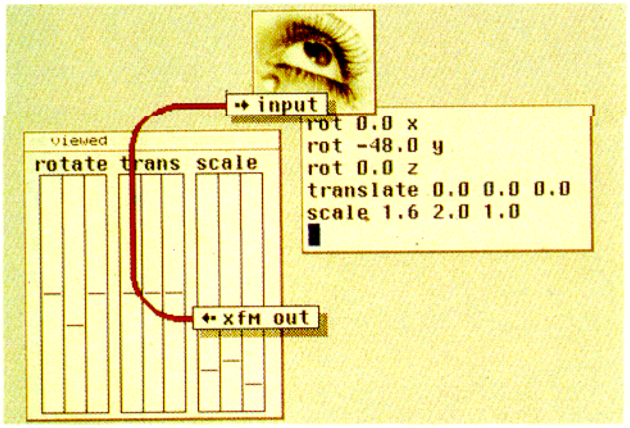
- *tape*. The tape recorder has one input port and one output port. By interacting with a menu we can erase the recording, start recording, or play back what has been recorded. While the recorder is in record mode, new objects are saved in a linked list as they arrive.
- *render*. The rendering component supports different rendering qualities from wire-frame to ray tracing. As input, it takes a description of a geometric object and a set of viewing transformations.

Now we will show how to combine these tools to interactively design swept surfaces. First, the user starts the components and connects them together as shown in Figure 2. The curve editor is connected to a sweep component. Two view editors are connected to the sweep component. One of these controls the view of the surface. The other provides a transformation that is iterated to create a swept surface. We can make a wide variety of surfaces in this way. For instance, a surface of extrusion is created by setting this to just translate in z, while a surface of rotation is made by setting the sliders to rotate in x or y.

There are two types of data that are being communicated between components in this application: short lists of transformations from the view editors, and descriptions of geometry from the curve editor and the sweep component. A typical output from one of the view editors is shown in Figure 3, while Figure 4 shows the output of the curve editor.

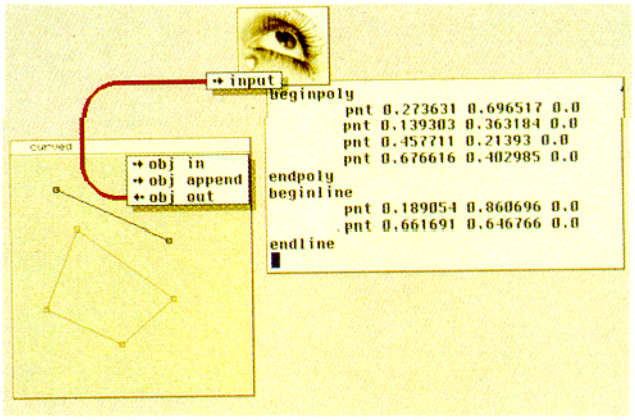
The user constructs this composite application by interacting with the connection manager. To do this, the components were created by making selections on a menu.

Next, the components were interconnected by displaying the terminals on each component, and drawing wires between them. This complete network was built in less than a minute.



```
rot 0.0 x
rot -48.0 y
rot 0.0 z
translate 0.0 0.0 0.0
scale 1.6 2.0 1.0
█
```

Figure 3. Output from a view editor.



```
beginpoly
        pnt 0.273631 0.696517 0.0
        pnt 0.139303 0.363184 0.0
        pnt 0.457711 0.21393 0.0
        pnt 0.676616 0.402985 0.0
endpoly
beginline
        pnt 0.189054 0.860696 0.0
        pnt 0.661691 0.646766 0.0
endline
█
```

Figure 4. Output from a curve editor.

Interacting with the Application

Every component provides a separate context for interaction: an interaction frame. The interaction frame of each component supports a visual representation and lets the user interact with this representation using the mouse. The window system allows the user to direct input events to a particular interaction frame.

The composite application consists of a population of interactive components. Each component reacts to messages that are received from the user or from other components. This reaction normally involves updating the visual representation and possibly sending a message out one of its output ports. Since components are dependent on each other, changes made while interacting with one component can propagate to other components.

In this example, interactive changes to the shape in two dimensions can propagate to the sweep component. This lets the user edit the shape in two dimensions and see the result in three dimensions. Interacting with the top view editor causes the sweep component to display the surface from different view points. In the same way, interaction with the lower view editor modifies the incremental transformation that is applied by the sweep component as it generates the surface.

Extending the Application

This composite application (sandwich) can easily be modified or extended. Suppose we want to create an animated set of views of the swept surface. This can be done by adding a component that acts as a tape recorder, as shown in Figure 5.

The tape recorder has one input port and one output port. By interacting with a menu we can erase the recording, start recording, or play back what has been recorded. While the recorder is in record mode, new objects are saved in a linked list as they arrive. Notice that the view editor output is connected to the recorder *and* the sweep component. Any output port can deliver data to any number of input ports. This lets us monitor the effect of the view editor as we record its output. After a series of objects has been recorded, the recording can be played back once or continuously in a loop. While the recorder is playing and the view of the surface is changing, we can interact with the shape of the surface or the sweep transformation and see the result in real time.

The recorder is a general purpose component that can be used to save and replay a sequence of objects whether they are viewing transformations or geometrical shapes. So we could also use a recorder to play a sequence of different shapes created by the curve editor.

If we decide we don't like using sliders to control our view of the swept surface, we can use different kind of view controller. Figure 6 shows how a hemispherical viewer can be added to the application. Notice that the output of *both* view editors are connected to the view input of the sweep component. The view of the swept surface will follow the sliders or the hemispherical view editor, depending on which component we interact with.

This demonstrates how different user interfaces may be bound to an application. It's equally easy to support multiple simultaneous user interfaces.

As a final example, let's create a shaded rendering of the swept surface. To do this, a general purpose rendering component is used with its own view editor. Figure 7 shows how the renderer has been connected.

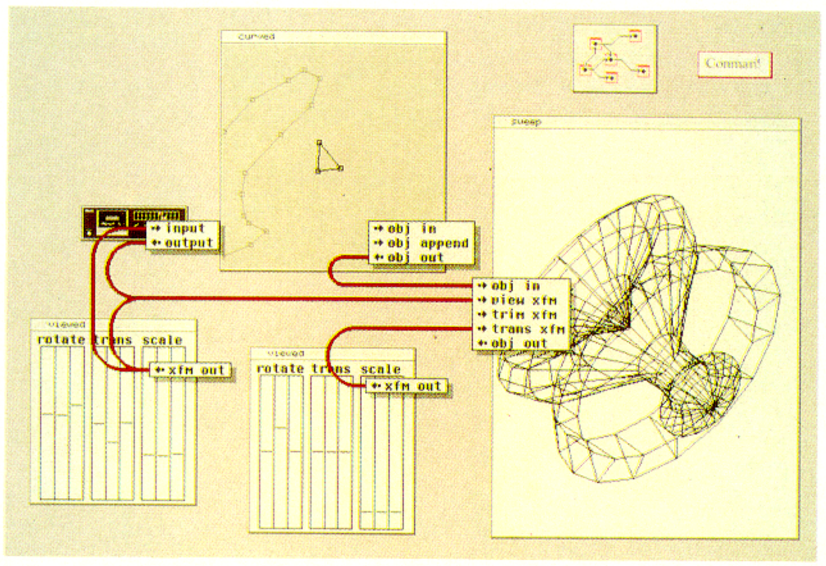


Figure 5. Adding a tape recorder.

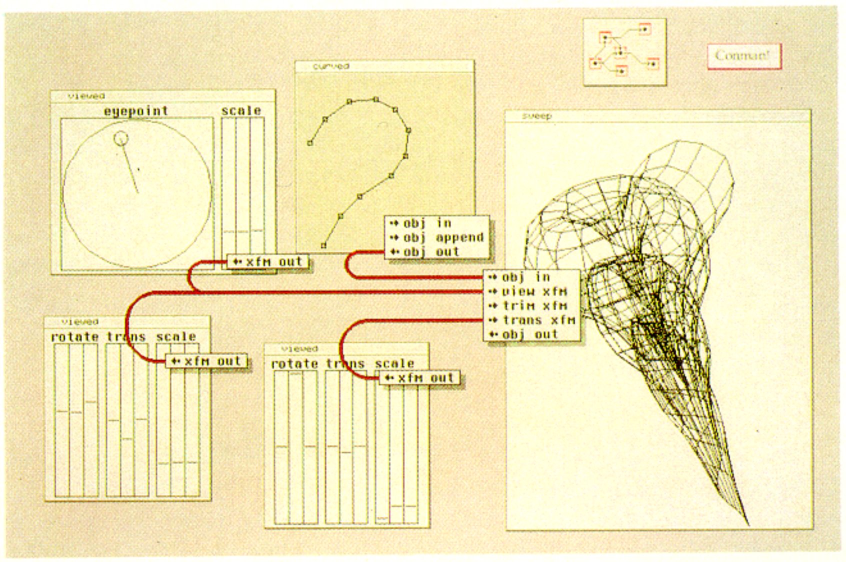


Figure 6. Using a Hemispherical view editor.

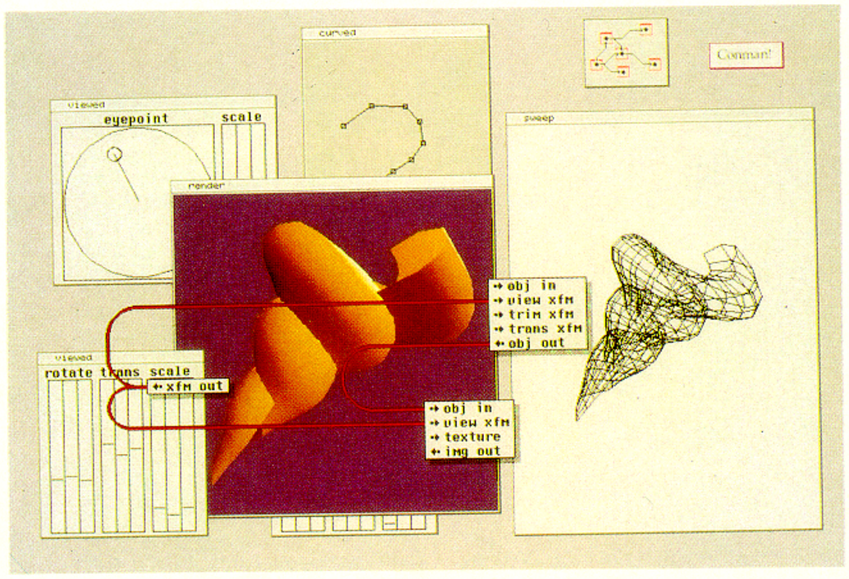


Figure 7. Connecting in a renderer.

Other Useful Data-Flow Components

Many other components have been developed for this data-flow environment. A watch component lets the user inspect data that is flowing across the screen, and a simple interface to the file system has also been developed. A mixer has two input ports and one output port. This component can be used to interpolate between two views, two shapes or two rgb colors. The mixer can also be used to concatenate the two inputs or randomly interpolate between individual components of the inputs.

A component called *tolines* converts an image into outline geometry. Figure 8 shows this component being used with sweep and render to make an extruded logo. A low-pass filter component can be placed between a view editor and another component to filter view transformations over time (See Figure 9). With this component in place, a sudden step translation will result in the geometric model moving along an exponential curve towards the new position in time. This kind of pseudo-dynamics gives the model a feeling of mass. The low-pass filter component is also competely generic - it can be applied to changing geometry as well.

A graftal plant [Smith 84] component accepts a gene description on one of its input ports, a leaf shape and view transform on other input ports. Figure 10 shows this composite application.

As a final illustration, figure 11 shows a paint component that gets the current drawing color from a simple color editor, and the brush shape from a curve editor. The curve editor output is connected to a component that transforms a geometric shape. This gives the user control over the scale and rotation of the brush.

Implementation

ConMan runs on the Silicon Graphics IRIS Workstation under the Mex window manager [Rhodes 85]. Each component process is programmed in the C programming language using the IRIS graphics library [Silicon 84] for graphic display. A detailed description of how this system is implemented can be found in [Haeberli 86].

The connection manager ConMan is a user process running under the window manager. Client components need to describe text labels for input and output ports. The user needs to be able to alter the interconnection of components.

When a client component starts up, it sends messages to ConMan indicating the input and output ports it uses, with a text string to label each port. The user can interact with the connection manager to add or delete connections between different ports on different components. The structure of the interconnection is maintained by the connection manager.

The graphics system supports an input queue to deliver events to each component. User, system and interprocess communication (IPC) events appear in this input queue. User events indicate changes in the mouse

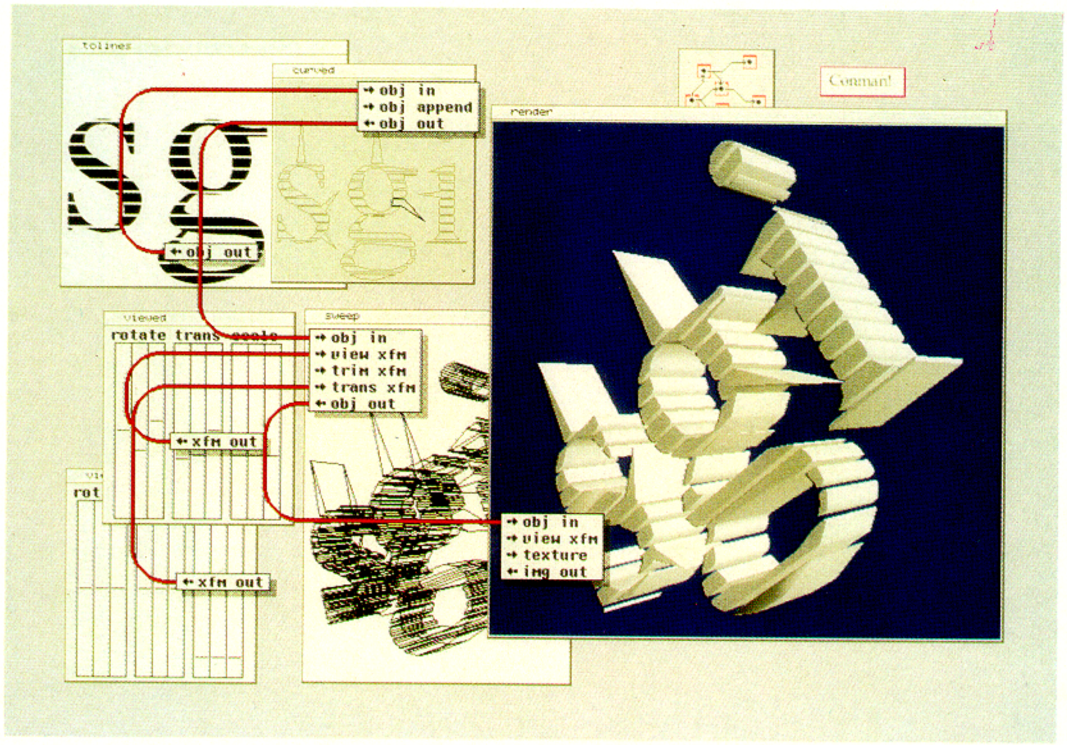


Figure 8. Extracting geometry from an image to make an extruded logo.

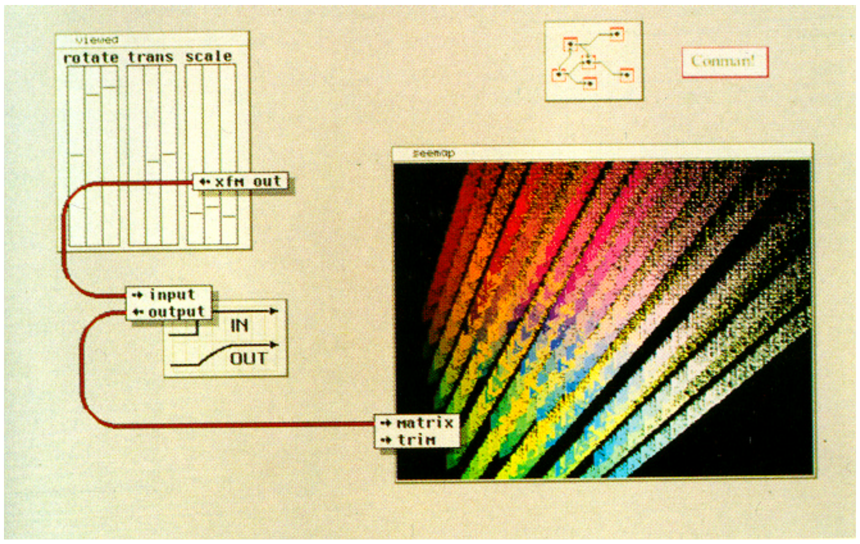


Figure 9. Using a low-pass filter for pseudo-dynamics.

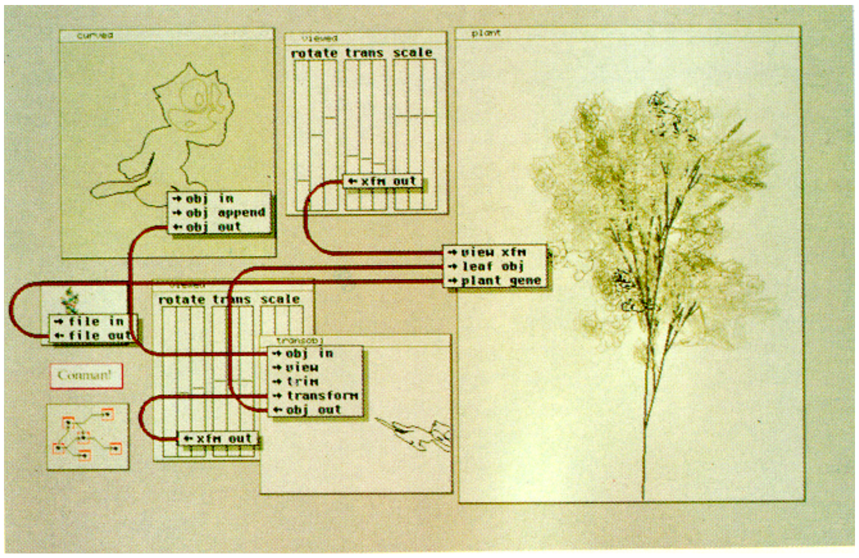


Figure 10. An application for graftal plant design.

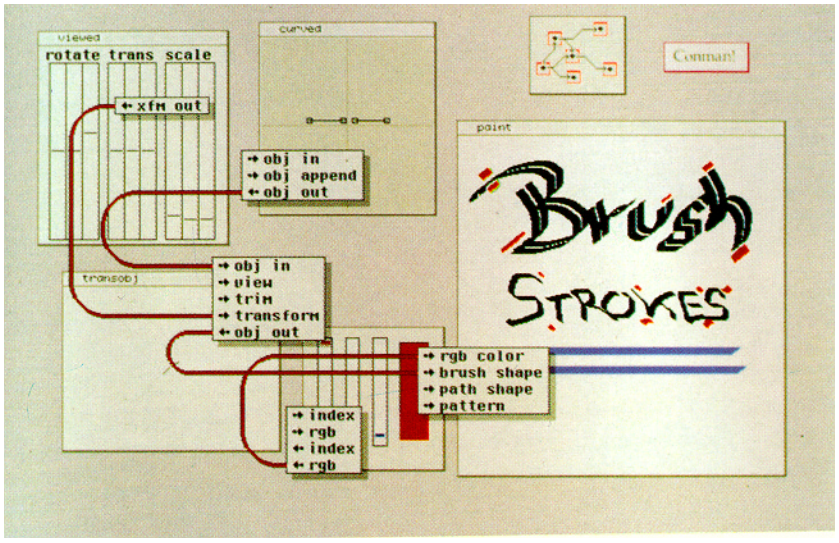


Figure 11. A paint application.

buttons, or position. System events notify a component that it should redraw because its window has received additional exposure. IPC events indicate that a message is available from another component.

Communication between components is accomplished by typed, variable sized, synchronous messages. To send a message, data is written into a file that is associated with the output port. Then the component notifies the system that there is new data available from this particular output port. This blocks the sender and places notification tokens in the input queues of all the components that are dependent on this output port. When a component receives an IPC event, it reads the message from the appropriate file, and explicitly replies.

In the current implementation, all data is transferred using a textual interchange format. System performance could be improved by using binary messages. The IPC mechanism described briefly above, using files and special system calls has recently been reimplemented to use standard UNIX sockets.

Conclusions

By providing graphical support for communicating sequential processes we create a primitive visual language that lets users interactively construct and modify applications on the fly. The connection manager lets the user create dynamic visual expressions out of interactive components.

Currently, the only data types (nouns) being transmitted between components are transformations, geometric shapes, RGB colors and bitmap images. We plan to extend the vocabulary by adding data types to describe text, fonts, and streams of input events. We also expect the vocabulary of data-flow components (verbs) to grow to support key frame animation, solid deformations and image processing.

ConMan has many implications for application developers and users of interactive workstations. Applications are really programmed at two distinct levels. A developer uses a conventional programming language at the component level. Both the user and the developer use a visual language at the level of the application.

Developers are encouraged to break monolithic applications into functional components that communicate with each other using high level data structures. Careful design of components makes them usable in many different contexts, and communication between applications is easy. ConMan promotes software modularity and healthy competition between components. For example, if a better view editor becomes available, it can easily be used by everyone. In this system, sharing of functionality happens at the component level.

Instead of supporting a single interaction frame with a single process, we use multiple processes in a windowed environment to provide multiple interaction frames, each with their own user interface state. An application is an orchestrated collection of interaction frames.

Components and the data passed between them form a vocabulary that is used to express the behavior of an application. This allows the user to explore the design space instead of being limited by the vision of the system implementors. The functionality of applications is open-ended.

Control over the application is returned to the user. Components of the user interface can be easily exchanged with each other. In this system, multiple simultaneous interaction techniques may be dynamically bound to an application. The functional binding of an application is completely dynamic.

That applications must be monolithic and self-contained is an illusion. We use the interactive medium itself to let the user design and extend applications.

Acknowledgement

I would like to thank Rob Myers for his continued enthusiasm for this project. Eric Brechner now at RPI implemented a version of ConMan that uses UNIX sockets for interprocess communication. Thanks to John Danskin at Digital Equipment Corporation in Palo Alto for allowing me to use their digital film recorder for the illustrations. Special thanks also to Dan Sears and Amy Smith for help with the manuscript.

References

[Blythe 86] David Blythe, John Kitamura, David Galloway and Martin Snelgrove, "Virtual Patch-Cords for the Katosizer", Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, 1986.

[Cardelli 85] Luca Cardelli, "Fragments of Behavior", Personal Communication, DEC Systems Research Center, Palo Alto, CA, 1985.

[Cardelli 85] Luca Cardelli, and Pike, R., "Squeak: a language for communicating with mice", Computer Graphics, 1985.

[Galloway 87] David Galloway, David Blythe and Martin Snelgrove, "Graphical CAD of Digital Filters", Proceedings of IEEE Conference on Computers, Communications, and Signal Processing, June 1987.

[Haeberli 86] Paul Haeberli, "A Data-Flow Manager for an Interactive Programming Environment", Proceedings of Usenix Summer Conference, 1986.

[Heckbert 87] Paul S. Heckbert, "Ray Tracing Jell-O® Brand Gelatin", Computer Graphics, 1987. [Hoare 78] C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM 21(8), August 1978.

[Kimura 86a] Takayuki Dan Kimura, "Determinancy of Hierarchical Dataflow Model", Technical Report WUSC-86-5, Department of Computer Science, Washington University, March 1986.

[Kimura 86b] Takayuki Dan Kimura, Julie W. Choi, and Jane M. Mack, "A Visual Programming Language for Keyboardless Programming", Technical Report WUSC-86-6, Department of Computer Science, Washington University, June 1986.

[Myers 86] Brad A. Myers, "What are Visual Programming, Programming by Example, and Program Visualization?", Proceedings of Graphics Interface 1986.

[Rhodes 85] Rocky Rhodes, Paul Haeberli, and Kipp Hickman, "Mex - A Window Manager for the IRIS", Proceedings of Usenix Winter Conference, 1985.

[Schulert 85] Andrew J. Schulert, George T. Rogers and James A. Hamilton, "ADM - A Dialog Manager", Proceedings of SIGCHI 1985.

[Silicon 84] Silicon Graphics Inc., IRIS User's Guide, 1984.

[Smith 84] Alvy Ray Smith, "Plants, Graftals, and Formal Languages", Computer Graphics, 1984.

[Smith 86] Randal. B. Smith, "The Alternate Reality Kit: An Environment for Creating Interactive Simulations." Proceedings of the IEEE Computer Society Workshop on Visual Languages, 1986.

[Tanner 86] Peter B. Tanner, Stephen A. MacKay, Darlene A. Stewart, and Marceli Wein, "A Multitasking Switchboard Approach to User Interface Management", Computer Graphics, 1986.