A Brief Overview of the VIOLA Engine, and its Applications

Viola is a tool for the development and support of visual interactive media applications. Possible viola applications range from a simple clock to a World Wide Web hypermedia browser (ViolaWWW).

ViolaWWW is what most people equate with "Viola", which is convenient. But it is important to keep in mind that ViolaWWW (the Viola–based World Wide Web browser) is just an application, albeit the most significant, of the Viola toolkit/language system, and that Viola can be used to build many other applications.

This paper will briefly gloss over the basics of the Viola engine, then briefly describe some Viola applications. The first section is a little thick with technical material, so if you're more interested in what Viola can do, read the Applications section first.

At the basic lavel, the Viola system is a combination of the following major subsytems:

- Object orientation support. Data is encapsulated into "object" units, and there is a classing and inheritance system for the objects. This model encourages an application to be divided into discrete units, and helps to make applications more scalable and cleaner in design.
- A scripting language. It is used to program the behaviour of each viola objects.
- A graphical elements (user interface) toolkit. The "widgets" exist as classes in the Viola class hierarchy. The set of widgets implemented in Viola is similar to those found in graphical user interface toolkits like Xt, plus more unusual widgets such as HyperCard–like cards and invisible celopane buttons, and hypertext textfield.
- Supporting libraries. Such as the libraries for GIF, XPM, WWW.

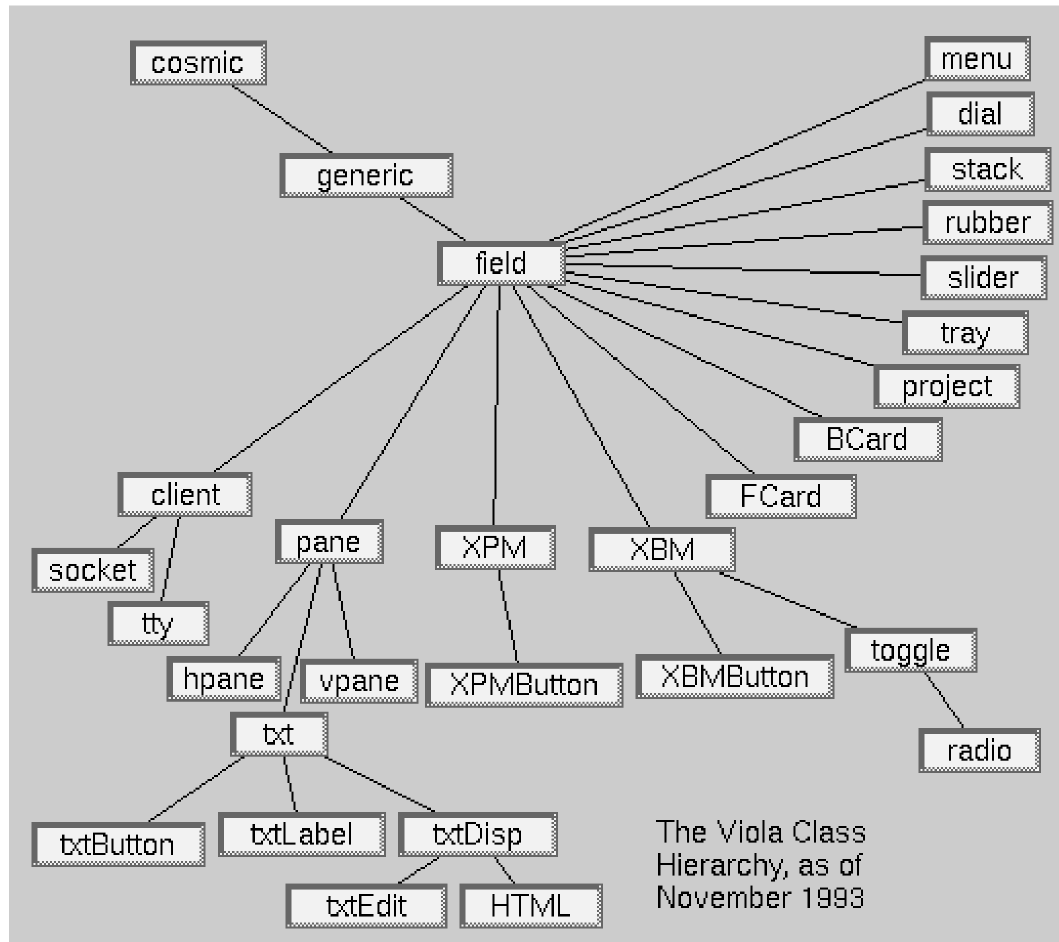
The Object Orientation Support –– Classes and Objects

The single inheritance classing model used in viola defines the basic types, or "classes", of object instances. Many of these predefined class types happen to be GUI oriented, because of the current application emphasis on hypermedia. But, many

classes are non−visual and have nothing to do with GUIs.

An object model is enforced to control complexity: to provide a relatively simple way of data encapsualization; for improving the size scalability and reusability of viola applications; and for network distribution of applications (distintive objects as addressable resources).

Shown below is the Viola class hierarchy tree. Note that the Cosmic class is the root of the hierarchy.



Each new class may define new attributes, and all classes inherit all attributes from the "super" (hierarchy ancestor) class. So, every object would have more or less the same sets of attributes, depending on the class of the object.

Among other things, this inheritance behaviour is useful for code sharing among the classes.

This class hierarchy may seem deficient compared to the GUIs provided by toolkits like Motif. But, it’s actually not as deficient as it seems. For the same reason as Tk/Tcl, Viola does not require the hard coding of, for example, dialog boxes, to achieve the same functionality.

Because of the interpretive nature of the system, many complex GUIs can be composed out of primitive elements, dynamically. To build a dialog box, a script could be written to create the necessary objects, and combine them together to constitute a dialog box.

To better show what a viola object is, consider this listing:

```
\class {txtButton}
\name {hello}
\label {Hello, world!}
\script {
        switch (arg[0]) {
        case "buttonRelease":
                bell();
        break;
        }
        usual();
}
\width {200}
\height {30}
\BGColor {grey45}
\BDColor {white}
\FGColor {white}
\
```

It is the file level representation of a Hello World "program" (just one object, in this case). The important to note is that an object is basically just a grouping of attributes, such as the object's class, name, script (the object's non–default behaviour), colors, dimension.

And this is what that object description instantiates to:

(If this were rendering on ViolaWWW, the button could actually be live on this document page!)

This "Hello World" example is obviously a very simple viola application that consists of only one object. A significant viola application, such as the ViolaWWW for example, consists of about 400 objects–– around 150 core objects that make up the WWW browser, and about 250 supporting objects that are specific to WWW data types such as HTML, etc.

Messaging System

Viola is very much message driven. Messages may be generated by a number of

ways. A message is typically caused by the user interacting with a graphical user interface object, but it could also be generated by other objects, or by a timer facility. Through a communications facility such as the socket, a message may also be generated from a remote source on the network.

In the above ‘‘Hello, world!’’ example, when the button is clicked on, that button object "hello" will eventually receive a "buttonRelease" message. At that time, the object’s script will be executed, and according to the script, a bell will be sounded.

Almost all viola objects have such message handlers that catch the message it’s interested in. Upon receiving a message, an object typically does some work, and/or generate more messages for other objects.

If the object does not have any message handlers, the message will ‘‘fall thru’’ the object, and (by way of usual() method) the class default action will occur. In this case, for example, the class action for a text button in response to button press/release events is simply the press/depress visual effect.

But as you see in the script, we have trapped the "buttonRelease" message to make a sound, before letting the flow of control go to the usual action.

So, a typical viola application consists of a collection of objects interacting –– generating, receiving, and delegating messages –– with each other, and with the user.

The Extension Scripting Language

As apparent in the example above, the viola scripting language is C–like in syntax. The language supports very few constructs, such as if, while, do, for, switch.

The commands like print(), exit(), create(), etc are all implemented as methods. Instead of building even these commonly used functions into the language grammar, they actually are just defined early enough in the class hierarchy so as to be accessible by all subclasses that may need them.

All objects can be individually programmed using the scripting language. Each object is essentially its own interpretive environment, and each object is its own variable scope. So, an object only affects its own variable values and can not directly affect other objects’ variables, thus minimizing side effects.

Having an encapsulated mini environment is useful not only for organizational purposes, but is also useful for enforcing security. This is particularly relevant in the Internet scaled World Wide Web context.

The ViolaWWW browser application has provisions for treating viola application files just as any WWW document–– transport via HTTP, and render mini viola

applications as if they're any web data. This is very powerful, but also raises security concerns.

So, to protect the user's system from any possibly dangerous scripts, ViolaWWW will mark any objects that is imported from remote sources to be "untrusted". In this way, an imported object may do whatever it's programmed to do within its own little environment without being able to adversely affect other environment. Also, since viola interpreter would then be able to distinquish the "untrusted" objects from the native "trusted" objects, it can limit the untrusted objects' operating system priviledges accordingly (ie: the interpreter will disallow untrusted object from invoking file system and other potentially sensitive methods or sub interpreters).

For optimization, object scripts are compiled into bytecodes before applying the bytecode interpreter on them. Because an object's script is basically a message event handler that is likely to receive many messages in its instance life time, the one time cost of parsing and simple transformation into bytecodes seems worthwhile. The gain in execution speed seems especially apparent when the objects deal with time critical mouse movement messages, or if there are tight looping operations. For further optimization, specific significant platform ports could conceivable have the bytecode compiler generate native machine code rather than the portable bytecode.

Applications

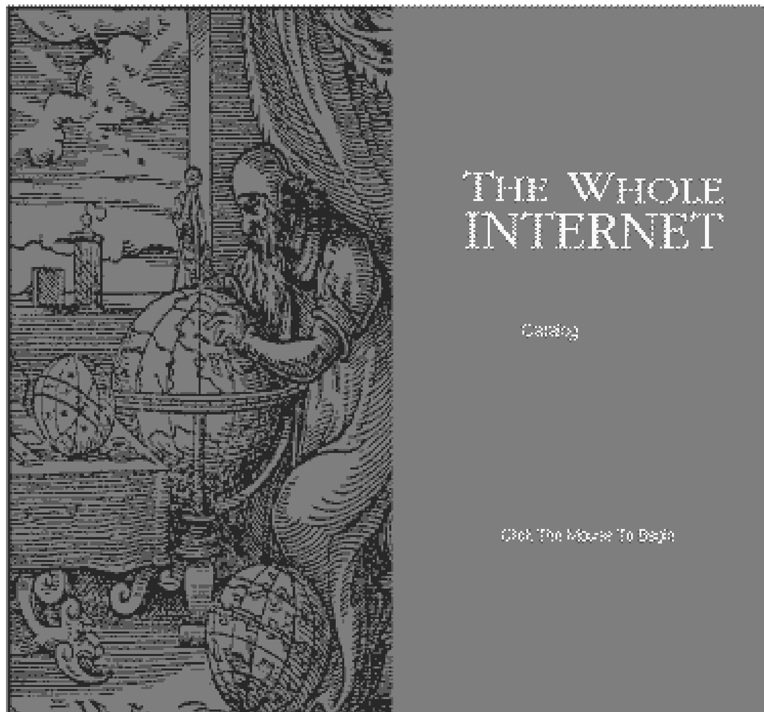
Along side the development of the Viola language/toolkit engine itself, there is also the development of working applications using the engine. The two development processes provide reality check for each other.

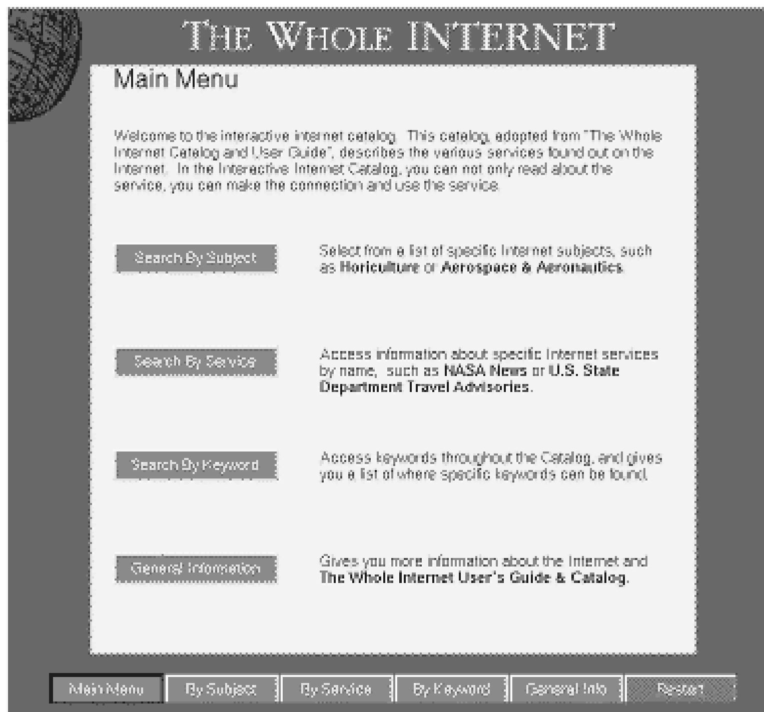
The Whole Internet Resource Catalog

An early application of viola: an electronic version of the resource catalog portion of the book The Whole Internet.

This application uses HyperCard style card−flipping technique to flip among four basic screen layouts. As it was intended for a kiosk type setting, the user interface is very specific and limited (compared to a general hypertext browser). But, the rendering code, the objects responsible for the page display, is shared by this application and the ViolaWWW browser application. Basically, just the user interface shell is different.

It's interesting to note that this application was developed with HMML (pre HTML+), and at the time a binary file format of HMML was used. This method has been abandoned, however, atleast for now.





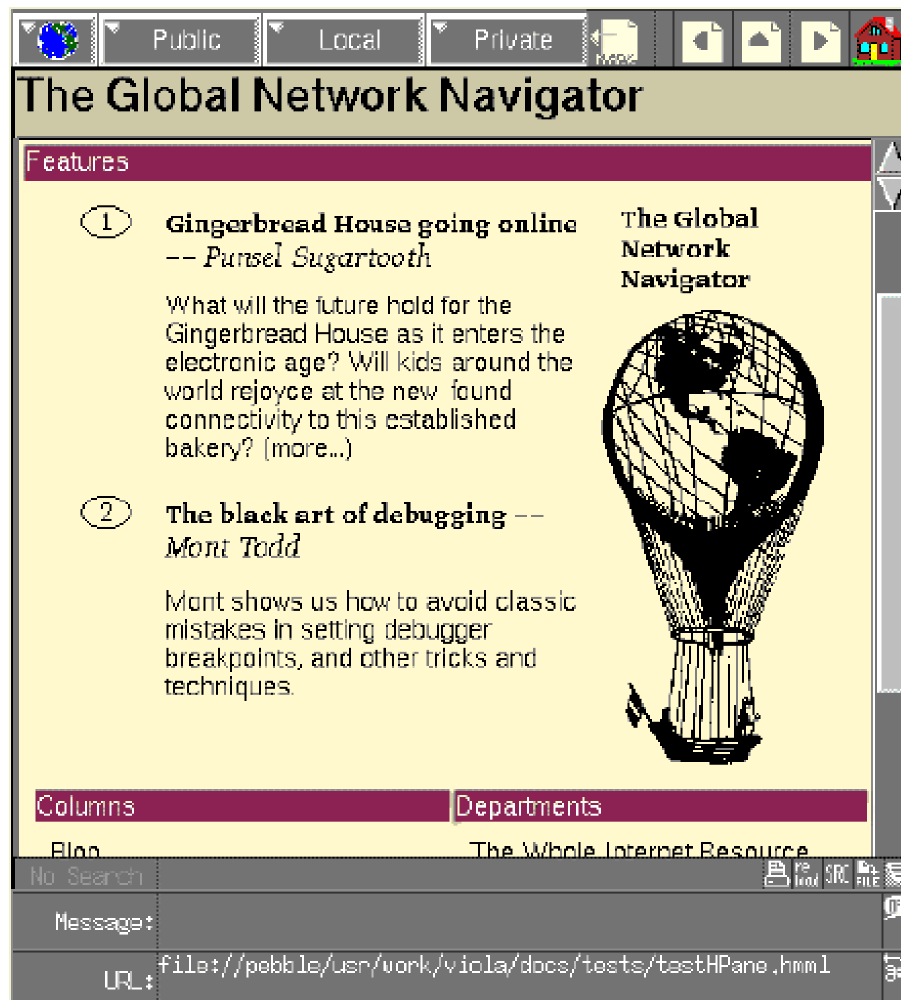
The Book Browser

This is an electronic book navigator application that works in conjunction with a document renderer (ViolaWWW). This application is basically a Table Of Contents navigator that lets the user browse the section titles, and selecting a title causes ViolaWWW to render the corresponding section page.

The browser/navigator also has a front–end to WAIS for searching within the book.

[Screendump to be inserted here once this application is revived once again]

World Wide Web Browser



[Note: the GUI design is of 1992 origin, but the rendering capability shown is about ’93]

The ‘‘ViolaWWW’’ application is the first X Windows based World Wide Web hypertext browser made available to the WWW community (mid 1992).

Since then, work on ViolaWWW has been on the next generation HTML (HTML 3.0). The latest (1994) generation of the ViolaWWW browser handles the standard HTML, including input–forms, tables, and some very rudimentary math equation support. In addition to the standard HTML 3.0, the latest ViolaWWW has added some extensions for greater (than the current generation of Web browsers) formatting capabilities and for embedding programmable objects in documents. And, it now also comes with an optional version that uses the Motif toolkit for the front–end.

Here are some snapshots showing some formatting capablities in ViolaWWW:

Tables

After the HTML+ RFC document:

	average		other category
	height	weight	
males	1.9	.003	yyy
females	1.7	.002	xxx

Figure 1: An Example of a Table

Collapsible/Expandable Lists

Clicking on the folder icon causes the list to expand or shrink. A listing on a page is no longer just static, but can be dynamic.

W3 Designs

- List item 1
- List item 2
- HTML Stuff
  - Stuff a.
  - Stuff b.
- HTTP Stuff
- More Stuff
  - Stuff.
  - Stuffy.

LEGO projects

Multiple Columns Formatting

One of the viola extensions to HTML provides richer formatting capabilities that is currently not available with other web browsers. The added <HPANE> tag is essentially a simple geometry management device that lets one format containers (ie: sections of text) side by side, nested, and can take max/min width constraint parameter.

The Journal of The Black Forest Programmers

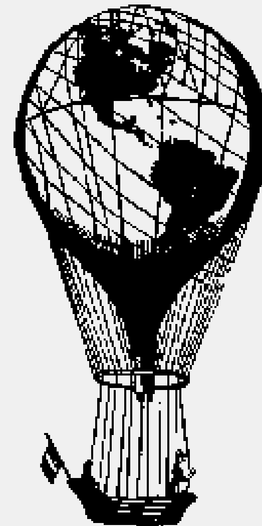
Features

Gingerbread House going online -- Punsel Sugartooth

What will the future hold for the Gingerbread House as it enters the electronic age? Will kids around the world rejoyce at the new found connectivity to this established bakery?

The black art of debugging -- Mont Todd

Mont shows us how to avoid classic mistakes in setting debugger breakpoints, and other tricks and techniques.



Columns

Bloo.

Blah.

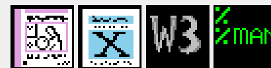
And a very fine Bleh.

Departments

UFO Sightings and Kidnapping Survivors Forum

Letters to Editors

Sponsors (actually, just random icons):

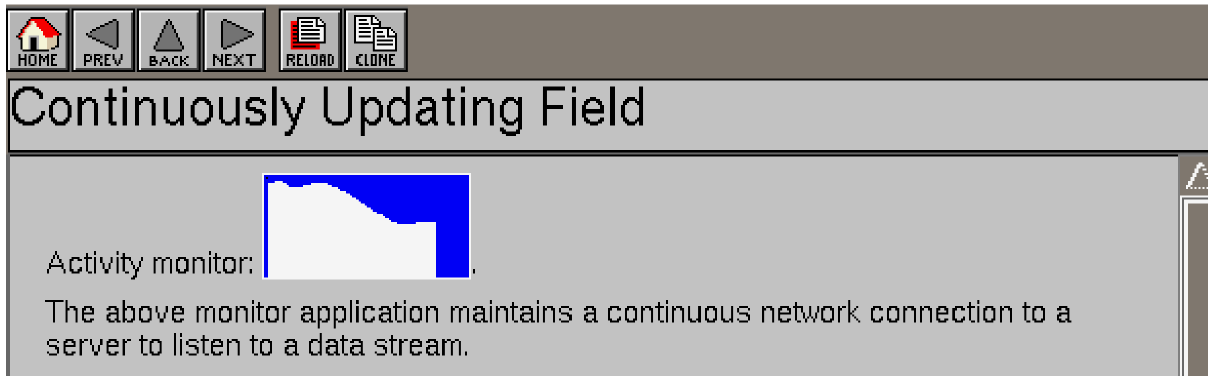


Embedding mini applications

Viola's language and toolkit allows ViolaWWW to render documents with embedded viola objects. Although the viola language is not part of the World Wide Web standard (yet?), having this capability provides a powerful extension mechanism to the basic HTML.

For example, if the HTML's input–forms does not do exactly what you want, you have the option to build a mini input–form application. And it could have special scripts to check for the validity of the entered data.

Or, if your document need to show data that is continuously updated, you could build a small application such as this which display the CPU load of a machine. Note that only the graph field is continuously updated, but not the rest of the document.



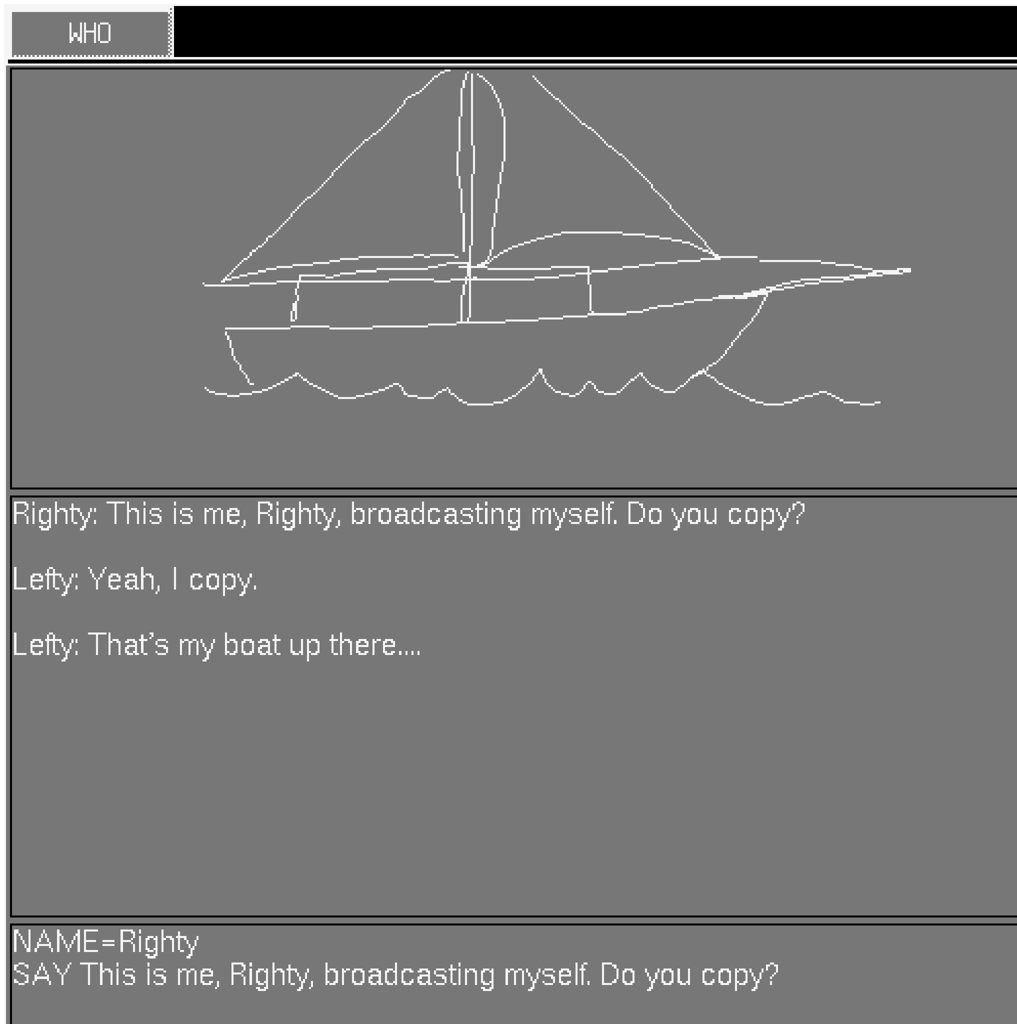
Other possible applications include front–ends to the stock market quotes, new wire updates, tele–video style service, etc.

Here's an example of a mini interactive application that is embedded into a HTML document. It's a chess board in which the chess pieces are actually active and movable. And, illegal moves can be checked and denied straight off by the intelligence of the scripts in the application. Given more work, this chess board application can front–end a chess server, connected to it using the socket facility in viola.

What follows is a demo of an embedded viola application that lets readers of the this HTML page communicate by typing or drawing. Like the chess board application

above, this chat application can stand−alone (and have nothing to do with the World Wide Web), or be embedded into a HTML document.

By the way, to make this possible, a multi−threaded/persistent server was written to act as a message relay (and to handle HTTP as well).



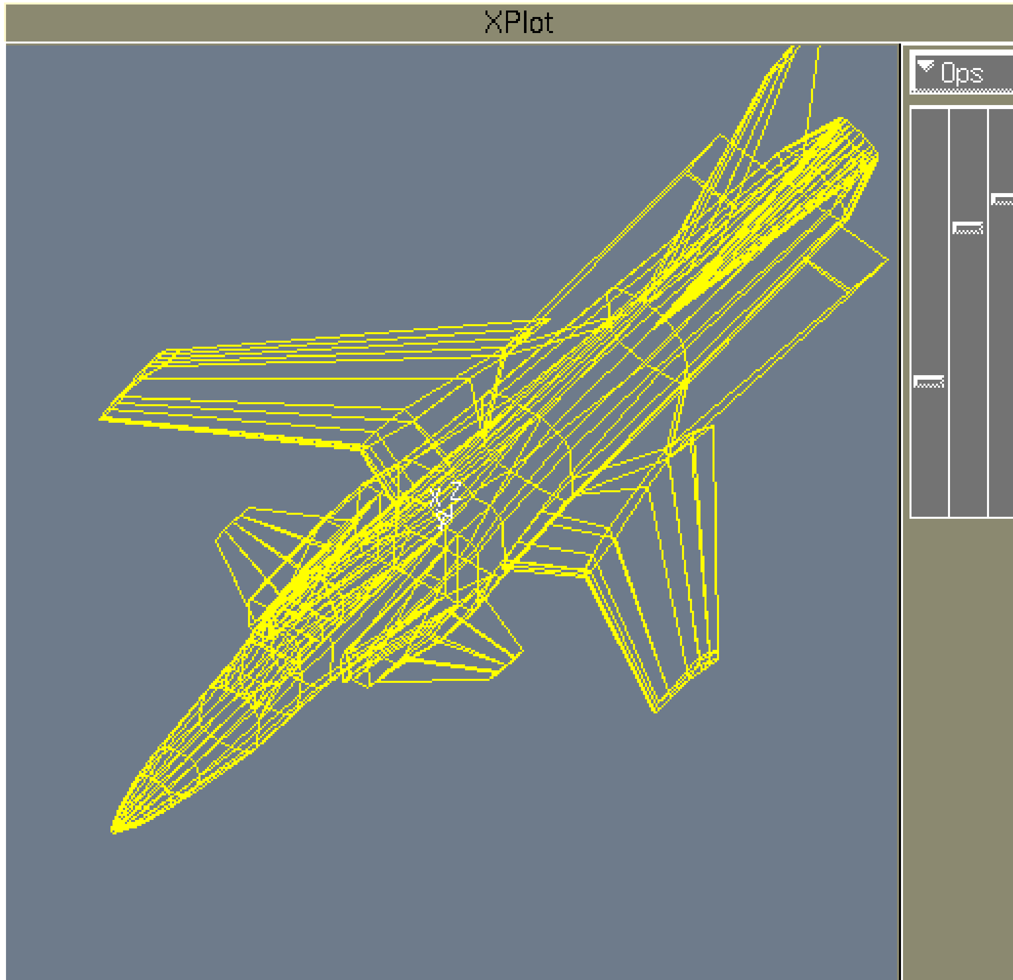
This next mini application front−ends a graphing process (on the same machine as the viola process). An important thing to note is that, like all the other document−embeddable mini applications shown, no special modification to the viola engine is required for ViolaWWW to support them. All the bindings are done via the viola language.

Put it another way, because of the scripting capability, the ViolaWWW browser has become very flexible, and can take on many new features dynamically. C−code patches and re−compilation of the browser can frequently be avoided.

This attribute can be very important for several reasons. It keeps the size of the core software small, yet can grow dynamically as less frequently used features are

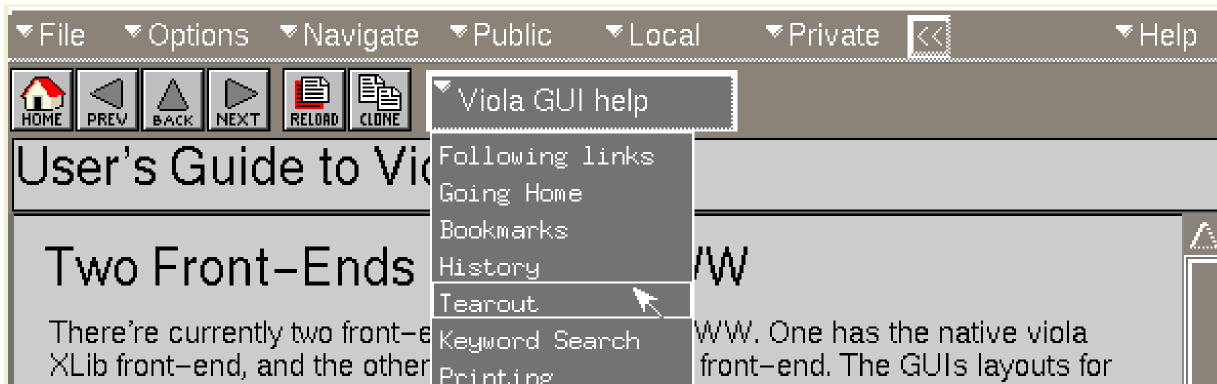
ocassionally used, or as new accessories/components are added.

Such new accessories can be as simple as little applets that accompany documents, or conceivable as complicated as a news or mail reader. An analogy is how Emacs's programming environment allows that text editor to become much more than just a text editor.



Not only can mini applications be embedded inside of documents, they can even be plugged into the ViolaWWW's "toolbar".

The following picture shows a "bookmark tool" that acts as a mini table of contents for the page. In this case, the bookmark is linked to the document (by using the <LINK> tag), and the bookmark will come and go with the document.



One can imagine many plug–in accessories/applets/tools possible with this facility. Like, a self guiding slideshow tool. Or, document set specific navigational tools/icons that are not pasted onto the page but are in a non scrolling place on the browser. Etc.

Summary

The Viola language/toolkit system provides an environment where applications are composed of groups of objects, where objects interact, by message passing, with the user and with each other.

As more applications are developed, more reusable objects will be created. And development of successive applications will become easier and easier. One of the goals of the Viola project is to accumulate a collection of objects useful for constructing hypermedia applications.

The immediate future direction of Viola development will continue to aim towards the path of hypermedia applications, with the World Wide Web as the document/object network transport infrastructure.

Pei Y. Wei (pei@ora.com)

R&D, Digital Media Group

O'Reilly & Associates