The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

Lessons from the Neighborhood Viewer: Building Innovative Collaborative Applications

Alex Safonov, Douglas Perrin, Josepth Konstan,
John Carlis, John Riedl, and Robert Elde
University of Minnesota

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Lessons from the Neighborhood Viewer: Building Innovative Collaborative Applications in Tcl and Tk

Alex Safonov, Douglas Perrin, Joseph Konstan, John Carlis, John Riedl
Department of Computer Science
Robert Elde
Department of Cell Biology and Neuroanatomy
University of Minnesota
{safonov,dperrin,konstan,carlis,riedl}@cs.umn.edu, elde@med.umn.edu

Abstract

This paper discusses the development in Tk of a collaborative browser for scientific image databases. The browser, known as the "neighborhood viewer," allows groups of neuroscientists to explore systematically a large collection of brain images. The paper discusses the application, its development, and a set of lessons learned during development. In particular, it shows how constraints and distributed constraints simplified development, discusses the implementation of a wavelet-based image format, and draws lessons about engineering experimental, evolving systems.

This paper tells the story of a group of scientists working together to build an environment in which many people could work together to study the brain. There are many challenges for neuroscientists in brain research, but we focus on the lessons that computer scientists learned as we used Tcl and Tk to invent new interfaces and applications. Our efforts bear fruit not only in the neighborhood viewer application (pictured in Figure 1) but also in a set of experiences and lessons that will make it easier for us to develop other innovative applications. We hope the reader will benefit from our experiences and thereby develop applications faster and more creatively as well.

We structure this paper as a development narrative. The next section gives the background and outlines the story. The following three sections focus more closely on three of the major lessons: lessons on the use of constraint programming, flexible coupling through distributed constraints, and the incorporation of wavelet-compressed images. The final section presents general lessons about engineering innovative and complex applications using Tcl and Tk.

The Neighborhood Viewer

Once upon a time... The context of this story is a larger, interdisciplinary project to extend database management system (DBMS) technology to support, in an integrated fashion, large volumes of neuroscience data from structural, functional, and behavioral studies using rodents. We intend to integrate neuroscience at the data level and thereby both enable cross-disciplinary neuroscience research and drive applied computer science research [1, 3]. While our initial focus was dealing with structural data (primarily images obtained from several different modalities, e.g., confocal, laser microscope and digital camera), it quickly became apparent that we had an opportunity to build novel user interfaces. This story describes their evolution.

As we proceeded, it became clear that no one of us was learned enough in both neuroscience and computing to develop these applications. Accordingly, our paradigm required two crucial elements:

mutual education We needed to educate each other. While computer scientists were already building scientific databases [7, 2], and neuroscientists were computer literate, there was much vocabulary to learn just so we could talk to each other. Also computer scientists needed to learn what neuroscience was about - issues addressed and ways of neuroscientific work. Notable was that neuroscience work changes as new imaging technology becomes available. Neuroscientists needed to learn ways of computer science work and to get a sense of the possible.

joint invention We worked together to invent interfaces. Several times we iterated through a two stage process. First, computer scientists observed neuroscientists doing neuroscience, and then jointly held brainstorming sessions. Second, the computer scientists designed and built a prototype, brainstormed on the prototype's interface details among themselves, and then jointly examined it with the neuroscientists. All were stimulated to invent new interface features.

These elements occurred simultaneously. Indeed, we did not begin to truly understand each other until our minds met at a prototype's interface. Note that we did not prototype as a means of users transmitting known, but unarticulated requirements. What we did was invent new tools, and, thus, new ways to do neuroscience. Note also that a) we did not try to have each prototype support every valuable feature, b) while eventually much of our data will be stored by a DBMS, we did not insist on using the DBMS for prototyping, and c) the later iterations described next really did overlap somewhat, but the story is simpler to tell if we ignore that.

In the first iteration computer scientists found out that confocal images a) are captured in modest-sized rectangles (about 500x750 pixels) applying 1 to 3 wavelength filters; b) are, in a labor intensive fashion, beginning to be "montaged" into larger rectangles. Confocal microscopy allows to capture many images at varying depths (z values) from one specimen. We characterized the data as "locally dense, but globally sparse" - most neuroscientific work focuses on a few "named brain locations." Neuroscientists are interested in the spatial juxtaposition of, for example, opioid acceptors and acceptors, each appearing under a different filter. A simple viewer was built that had straightforward pan and zoom features, a bookmark feature, and, mimicking what happens at the microscope, could "flicker" between the part of an image captured with different filters. Together we decided that:

- Monitor real estate is precious and that the interface should allow the user to toggle off temporarily unneeded buttons, sliders, etc.
- One needs to be able to go forward or back through a stack of images, which we termed "rolodexing."
- While pan and zoom are needed and good for answering the question "what is this?", they are insufficient for the question "where am I?"A zoom out followed by a directed zoom in does work for knowing where one is within an x,y image, but is not sufficient for knowing where one is within the brain in the (relative) z dimension.
- It is a labor intensive and expensive task to print out an image, mark interesting spots and record comments. Doing these tasks on-line became a requirement.

It was not hard for us to decide at that point to implement in Tcl/Tk, because a) we were already comfortable with Tcl/Tk, and b) we did not know where the interface would evolve.

In the second iteration computer scientists learned that images were generally from a coronal cutting plane, but that sagittal and horizontal cutting planes (respectively: vertical - front to back; vertical - side to side; and horizontal - top to bottom) are also valuable. They observed neuroscientists, to help themselves determine "where am I," using a rat brain atlas consisting of coronal images with a visual index: a sagittal silhouette plus a "here" vertical line. Computer scientists saw expert neuroscience researchers using the atlas much less than graduate students. Together we decided that:

- Users should be able to simultaneously display related images from stacks in different cutting planes - generalizing the altas's silhouette. The intersection point of the image planes defines a "neighborhood."
- Movement (pan or rolodex) in one image causes a redefinition of the neighborhood and should induce appropriate "yoked" movement in other images.
- To aid in answering "where am I?" users should have the option to display surrounding, "near-by" images of the current focal image in a stack.
- For yoked movement to occur users must a priori assert how far apart in (relative) z images within a stack are, and how pairs of stacks line up.
- Users must be able to temporarily unyoke and then reyoke.

As we added more and varied interface elements, the computer scientists found themselves bogged down. Even with Tcl/Tk our code was becoming too intricate, and implementing got progressively more difficult. This motivated us to consider the constraint model.

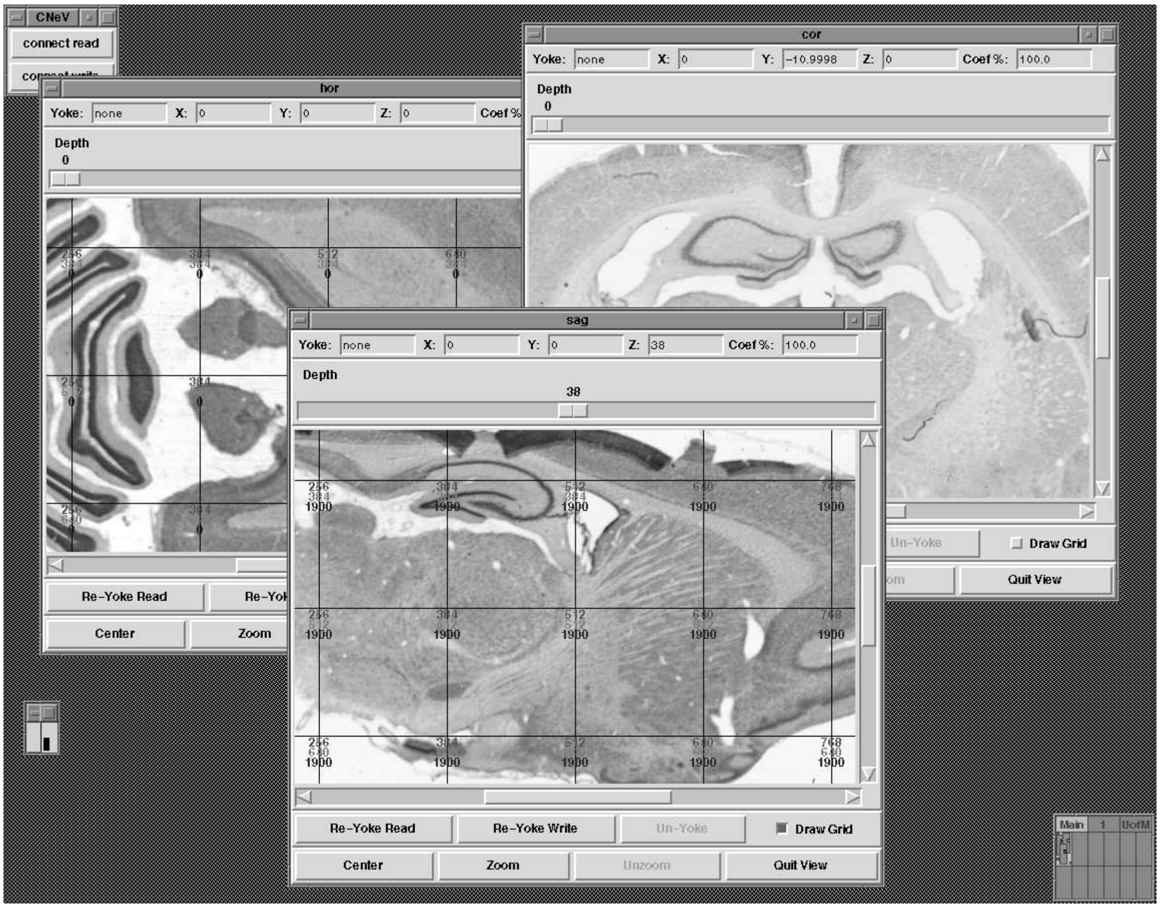

Figure 1: The neighborhood viewer. The three brain images are the coronal, horizontal and sagittal views. Each view has yoke controls, pan/zoom controls, and separate x, y, and z coordinates.

In the third iteration computer scientists observed neuroscientists laying out a handful of printed images of the hippocampus of subject rats. Neuroscientists looked at the side-by-side images and discovered things they could not see before. At the same time, computer scientists observed neuroscientists struggling to work with distant collaborators. Packages, phone calls and email were slow and thin communication channels. Together we decided that:

- The neighborhood viewer needed to be able to display an arbitrary number of stacks.
- A "storyboard" viewer (barely begun) was needed to allow the neuroscientists to position various images for scrutiny. The arrows and annotations developed for the single image viewer applied here too. A storyboard could serve as a visual index into the image database.
- Since monitor real estate is precious, provide for multi-machine viewers.
- Provide for a) the simultaneous viewing of neighborhoods and storyboards by distant (or local) collaborators, b) their viewing asynchronously, and c) audio communication: real-time, recording, and conversion to text. (b and c are future work.)

Having had some success in building the neighborhood viewer using a constraint model, it was easy for us to decide to try a distributed constraints model to satisfy the multiple machine and collaboration requirements.

In the fourth iteration, spurred on by bottlenecks caused by handling multiple displays of large images, computer scientists discovered that neuroscientists did not need fully precise images to determine "where am I" but to explore "what is this?" they required detail. Together we decided to:

- Display, at first, only fuzzy images, which are sharpened on demand. (This can be done relatively quickly)
- Allow a user to see nearby images at the same or fuzzier precision.

Image processing costs led us to search for an alternate solution to these two requirements. Wavelet analysis offered a solution which met both. We found that as little as .2% of the data is sufficient for coarse "where am I" decisions, while 2% serves for finer ones. For most "what is this" questions 5-10% often works and only occasionally is full detail needed.

In the next sections we present lessons learned in using constraints, distributed constraints and wavelets, overall lessons, and conclusions.

Programming with Constraints

Imagine a neuroscientist browsing brain data. Multiple 2D views of data are displayed; each is a projection centered on the same point in the brain. As the neuroscientist navigates in any of the views, the other views should update their images to remain consistent, i.e., centered on the same point. No matter how navigation is done - by dragging the image, scrolling, or directly entering new center coordinates, all views must remain consistent. Imagine also that the neuroscientist wants to display in the horizontal view two additional images that are 10 microns above and below the focal image. In our implementation of such a multi-view browser we would like to avoid two problems: having each view explicitly update all other ones, and remember which views are added and deleted.

The constraint-based programming model offers a good solution. This model allows the programmer to specify relationships between variables explicitly, independent of the control flow. The constraint engine enforces these relationships at runtime by updating dependent variables whenever related source variables change. These relationships are called formulas or constraints.

Our original implementation did not use constraints. Scrollbars were used to pan in x and y directions, and a scale allowed users to select another image from the stack. When moved, a scrollbar and scale widget sent the changed center coordinates to all scrollbars and scales in other views. We found our code hard to read and modify. We therefore chose to use TclProp [5] to re-implement the neighborhood viewer. With constraints, we identified a global 3D position and encapsulated each view's state in the `x`, `y` and `z` coordinates of its center. Since each view is the projection of the 3D space, we found it intuitive to express the `x`, `y` and `z` coordinates of its center as transformations of the global 3D position. The differences in data communication and Tcl code were significant and are illustrated in Figure 2.

The encapsulation of the view coordinates gave us the two important benefits. When the neuroscientist navigates in a view, for example, by moving a scrollbar, the scrollbar just sets the coordinates of the center appropriately. The formulas ensure that global position, and variables storing center coordinates of other views, are updated. When a view's coordinate variables change as a result of formula propagation, the constraints linking them to control widgets are triggered, and update the image in the view. For example, constraints on `x` and `y` of the view center scroll the view window in the `x` or `y` direction, respectively. The constraint on `z` loads a new image corresponding to the new value of `z`. The code became a lot simpler, since coordinate transformations were encapsulated in formulas between local and global coordinate variables, and did not have to be specified for each scrollbar or scale widget.

Second, with constraints, widget callbacks did not have to know about widgets in other views. Scrollbars and scales communicate via coordinate variables related by formulas. We could now add a view from a new direction without going through the program and adding move-notification code to each scrollbar or scale callback. The code became easier to modify. Views were originally separate modules for each projection plane, with a large amount of code overlap. Now we have merged them into a single view module, customized by declaring appropriate formulas linking local and global coordinate variables. Altogether, by adopting the constraints model we reduced the number of lines of code by about a factor of two, and made the code a lot more understandable and extensible.

More generally, we learned two important lessons about constraints:

Constraints make code simpler. If a variable is updated in several places in the program, we only need to specify once what happens after the update. During the development of the neighborhood viewer, the neuroscientists asked to implement saving position bookmarks. When the neuroscientist

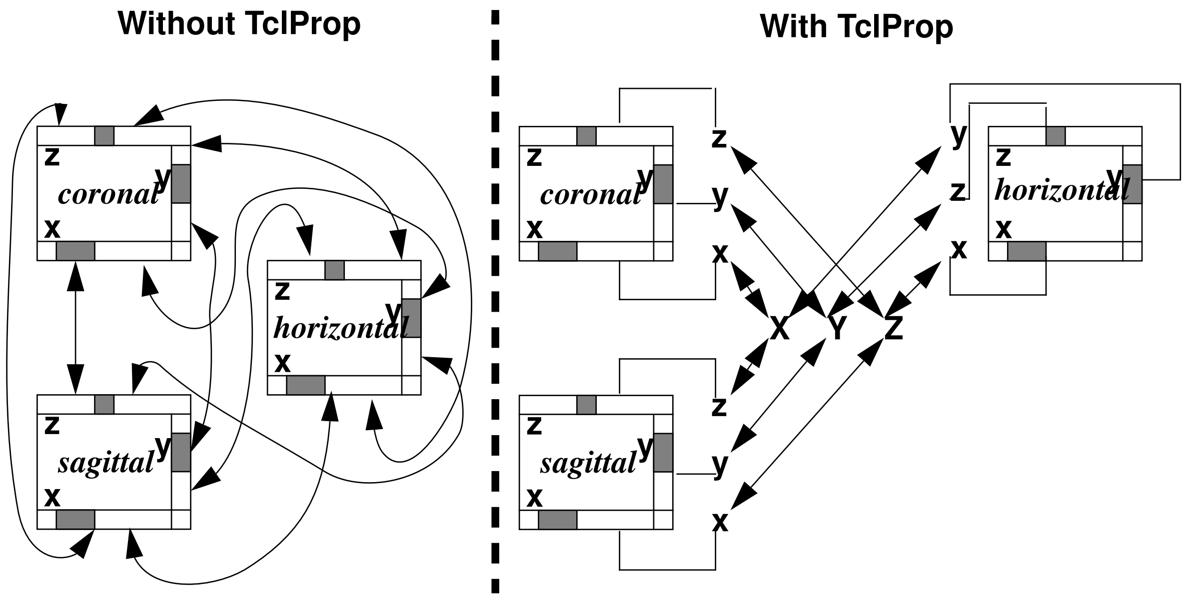

Figure 2: Two implementations of the neighborhood viewer. In the original implementation, each control widget must be aware of all other ones. In the implementation with TclProp, each control widget is linked via a constraint to a `x`, `y`, or `z` coordinate of the view center. The latter are in turn linked to the global position. Both links are two-way. Adding new views with TclProp requires only establishing connections between the center coordinates of the new view, and the position variable. Without TclProp, adding new views requires updating the code for each control widget of the other views.

goes to a saved bookmark, all views should center on the point with the global coordinates stored in the bookmark. If we use constraints to relate the views' centers to a global position, we just need to implement the bookmark interface and set the global position when a bookmark is selected; the constraint engine takes care of notifying the views of the new position. This localization of the update propagation code inside constraints is what makes the code simpler and more readable.

Constraints make code easier to modify and extend. Consider the task of adding a view from another stack of images. In the traditional model, we would have to add code to each existing view that notifies the new view of the change in position. With declarative programming, we create formulas that relate the view's center coordinates to the global position coordinates. Views are unaware of one another's identity, communicating only through the constrained coordinate variables. Constraints thus allow us to focus on the part of the application being developed, while relying on the constraint engine to maintain consistency between variables.

The declarative programming model also creates new challenges to the programmer. In Tcl, constraints can be declared and executed in different contexts, so care must be taken about variable substitutions. We sometimes found it difficult to determine the program state during execution, since we had no direct control over when and how variable updates were propagated. We added extensions to TclProp to dynamically disable and enable constraints using several criteria; these extensions were useful for determining the state of the running program. We think there is a need for a "constraint visualizer" to help programmers evaluate and modify execution of a constraint-based program.

In sum, constraints are generally useful where relationships between program variables can be expressed independently of the program flow, and where there are multiple sources of variable updates. The declarative model is especially appropriate for event-driven programs, which do not have a predefined execution path, and usually have multiple ways to manipulate the program variables. The benefits of constraints are code reduction and improved extensibility, due to localized update propagation and hiding the information about dependent variables inside the constraints. Networks of constraints can be easily changed at development or runtime. We believe that the benefits of the declarative model need to be further explored by extending constraint packages, investigating domain areas that yield themselves to declarative programming, and writing more constraint-based applications.

Distributed Constraints

Imagine a group of neuroscientists in different locations, browsing through the same brain data set. As they work together, they need all their data views centered on the same position in the brain, and need to have their views updated as any of them changes. Each neuroscientist may also disconnect from the group, browse independently, then reconnect and either move to the global position, or request others to move to her position. Normally, any participant should be able to move everybody's views; however, a chairperson can be selected and made the only user allowed to control navigation. The collaborating neuroscientists thus need to control the amount and type of their interaction.

This principle, known as *flexible coupling* [4], is common to many collaborative applications. Users must be able to dynamically connect to a session, disconnect, work independently, then reconnect. GroupKit [6] defines a rich API to manage sessions, among other things. Different collaborative applications require different degrees of synchronization and consistency. We can meet these requirements with flexible update propagation: variable updates may propagate immediately, after being validated by the application, or only when the user explicitly commits them. Depending on the number of participants and the formality of collaboration, different access control methods at variable granularity are needed. Examples are floor control, locking individual variables, and locking groups of related variables.

The constraints provided by TclProp, while effective for simple coordination of multiple linked views, are not powerful enough to implement flexible coupling of multiple users on multiple machines. To address the demands of collaboration, we needed to develop *distributed constraints*.

Distributed constraints provide a good framework for implementing flexible coupling for collaborative applications. We developed a distributed constraints module by extending TclProp to handle network communication using Tcl-DP [8], and implemented some features of flexible coupling. Dynamic disconnection and reconnection are supported by disabling and re-enabling constraints on variables of the specified view. The shared floor-holder token provides the floor control policy. Flexible update propagation is currently supported in the application, not at the toolkit level. We are currently examining ways to integrate flexible coupling into a distributed constraint module based on TclProp, and we plan to release that module at a future date.

We benefited from using distributed constraints in several ways. We were able to allow collaboration between geographically distributed neuroscientists, without significantly increasing code complexity compared to the single-user, multi-view version. As local constraints maintain views in the single-user prototype centered on the same point, distributed constraints ensure that all collaborating users see the same brain data. We achieved this by declaring distributed constraints between the coordinate positions of individual users, and the global group position. For flexibility, these constraints added a level of indirection–each user now has a "personal" coordinate that is linked by local constraints to the individual views. In this way, a user can systematically view the brain at an offset from the group view. For example, an individual neuroscientist may choose to view the brain 10 microns deeper (in the z-direction) than her colleagues, and at the same x and y coordinates. Her viewer would link to the global coordinate system as follows:

```
# link local x and y to globalX and globalY

DTP_inLink  $host $port globalX x
DTP_outLink $host $port globalX x

DTP_inLink  $host $port globalY y
DTP_outLink $host $port globalY y

# link local z to globalZ+10 and vice versa
```

```
DTP_inFormula  $host $port \
    z globalZ {expr $globalZ + 10}
DTP_outFormula $host $port \
    globalZ z {expr $z - 10}
```

Neuroscientists can disconnect from the collaborative session and navigate through the image data independently, while keeping their personal views of data consistent with one another. This is achieved by temporarily disabling constraints between the group position and user position, while retaining the constraints between the positions of this user's views. Similarly, they can still unyoke individual views from their personal coordinate system, whether connected to or disconnected from other users. Figure 3 shows how connection and disconnection are supported.

We are experimenting with other flexible coupling models, such as floor control, within the neighborhood viewer application, and we plan to move support for these models into our Distributed TclProp module as soon as possible. We are also investigating new models including group awareness through queries into the distributed constraint table.

Incidentally, the use of distributed constraints provided our users with a valuable application feature that would have otherwise been difficult to implement. With limited screen real estate and local image processing resources, it is convenient for a single user to use several workstations and monitors, each holding a different view or set of views. Our distributed constraint approach makes implementing this feature trivial, and also provides a mechanism for distributing pieces of images that are larger than a single screen across several monitors with linked scrolling. We expect this approach both to improve performance (by parallelizing expensive image loading) and to expand the horizon of interfaces that the neuroscientists can imagine.

To summarize, we developed a distributed constraint model based on TclProp [5] and Tcl-DP [8]. Our model supports a rapid design-code-invent cycle, implements some features of flexible coupling, and supports multiple, modifiable constraint configurations. By adopting the distributed constraint model, we achieved collaboration and local speedup in the NV while keeping the code readable and extensible.

We discovered, in the process of developing and using distributed constraints, that the key to flexibility is providing places in the constraint topology where links can be disconnected. It is important to have intermediate nodes in the graph representing users and views, to provide a site for local disconnection and transformation. Providing these intermediate nodes has not been very difficult and our observations have shown that the bottleneck of distributed constraints lies in network communication, not local constraint propagation. Therefore, any slowdown is negligible (indeed, the entire distributed constraint system performs comparably with RPC).

Finally, we believe that distributed constraints can be particularly beneficial for collaborative systems that require synchronous navigation through sets of spatial data. Examples of such application domains are medical and biological imaging, GIS, and collaborative data visualization in general.

Wavelet Lessons

Imagine a neuroscientist browsing brain data. Her two questions, "where am I?" and "what is this?", require different support. When answering the former question (or when zoomed out) not all of the detail stored in the database is needed. This situation provided us an opportunity to improve the responsiveness of our interface. We looked at reducing image transmission time by trading initial image detail for speed and, later, as it was needed or there was time, adding more detail to the image.

In order to trade off image quality for image loading speed, we needed a representation of the data that could be used to transmit an image in such a way that the quality of the image could be incrementally improved. That is, we want to incrementally improve an image's resolution, and do so at the user's command. We considered the interleaved GIF format. The popular WWW browser, Netscape, uses interleaved GIF to show the user a coarse view of a picture that is then refined with time. However, GIF was insufficient for our purposes because it can only be used to store 256-color images, and we wanted more flexibility.

Unfortunately, all of the Tk-supported image formats lacked the flexibility to store 24-bit color images in a form that can be efficiently loaded incrementally. So, we started looking at general requirements for our image model: rapid display of a good image and refinements to the image with minimal database access. If we found a suitable representation, we would see how hard it would be to extend Tk to support it.

We quickly focused on wavelet-based compressed

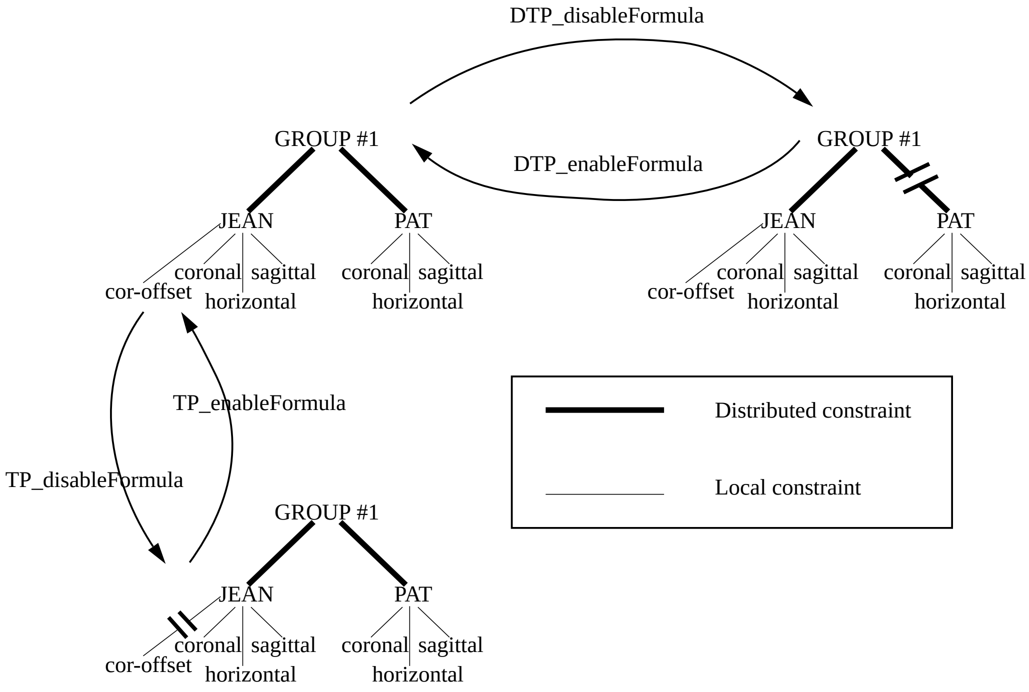

Figure 3: Flexible coupling in the neighborhood viewer. The constraints linking Jean's views with Pat's views can be disconnected at the user level (top-right) or at the view level (bottom-left).

images. For our purposes here we describe the wavelet transform simply as taking images (pixel values) and producing collections of coefficients. (Wavelets are a much more complicated topic.) These coefficients are averages and differences from the average over some area. One wavelet coefficient will represent the average intensity of the whole image whereas others contain information about smaller portions of the image. A network packet of pixels is just a set of dots, while a network packet of wavelet coefficients would generate a fuzzy but complete version of the original image.

To prepare for incremental resolution we a) transform the image to produce coefficients, b) sort the wavelets coefficients from coarsest details to finest, and c) store them. When the viewer requests an improvement in the image, more coefficients are read sequentially from disk. Sorting in this fashion satisfies our requirement that large improvements in quality occur early in the transfer. Indeed, just a few percent of the data is normally sufficient for a good image.

Wavelets work particularly well for the neighborhood viewer application. Work with early prototypes identified that users transitioned between two modes: positioning and studying. In the positioning mode, users are most interested in coarse shapes to navigate the large number of images in the database. Once the neuroscientist has found a particular region of interest, he will study it in detail. He now focuses on a small section of the image and is interested in seeing substantial detail. Indeed, given the expense of acquiring images, he will not tolerate any loss at this level. (Another benefit of wavelet transforms is that they are not inherently lossy, which is a decisive advantage in this application over the discrete cosine transform used in JPEG.)

Our interface evolved to incorporate a new operation: add resolution. Navigation now involves a sequence of "move, add resolution" cycles as the user moves closer and closer to the point of interest.

Wavelets appear to be the right solution. They provide the model of compression we sought, one that supports rapid display of good pictures and incremental display to reach original detail. Unfortunately, wavelets involve extensive calculation, and a back-of-the envelope estimate showed that wavelet code in Tcl would never perform satisfactorily. Being ecologically-minded, we sought out a library that we could reuse, Michael Hilton's wavelet library in C, made a few modifications, and created a wavelet-compressed image object in Tcl. Our original plan was to create a new image format, but since the current API for image formats does not support incremental image quality, we developed a "wavelet data" object that interacted with the standard photo image.

The resulting iteration of the neighborhood browser gave us a new performance trade-off. Wavelet-transformed images require less network transmission, but more computation. We find that the two versions are comparable on high-speed local networks, but that the wavelet model will become increasingly preferable as we consider transferring the images over slower, intercontinental networks. Furthermore, we are now poised to take advantage of any improvements in performance that become available for wavelet transforms (the "lifting" technique promises to double transform speed).

The lesson from this experience is that easily extendable systems make it possible to explore new technologies. Had we been developing in a system more rigid than Tcl/Tk, we probably would have abandoned the search for a better image format. Because of the clean interface for extending Tcl/Tk, we were able to integrate an existing wavelet library and explore more advanced interfaces.

We are currently working to extend our wavelet implementation in several ways. First, we are working on performance issues surrounding the inverse wavelet transform as these are the bottleneck for real-time applications. Second, we are examining techniques to handle images with widths and heights that are not powers of two. When we have addressed these issues, and defined a robust file format for wavelet data, we plan to make the wavelet extension available to others.

So, Who Gives a Rat's Brain?

While we hope the lessons we have discussed above help other developers as they build innovative applications in Tk, in many ways the more important message involves higher level principles of application design and development. In this section, we discuss some of the higher-level lessons that we learned, evaluate how well Tcl and Tk serve as development tools, and discuss some improvements.

Innovative applications do not have specifications. Developing the neighborhood browser was exciting because neither we nor our users knew what the application would become. It has been a long and continuing process for computer scientists to learn what is meaningful and useful to the neuroscientists and for neuroscientists to understand the range of options that can be provided. Most of our progress has been made in baby steps, rather than great leaps, because great leaps require too much understanding. It is easy for a computer scientist to say "oh, we'll build you a 3D visualization of the brain," but it takes time to understand why a 3D visualization is not as useful as the right combination of 2D images. Developing innovative applications and interfaces is an inherently iterative process, and the requirements and specifications evolve as development proceeds.

Turn-around *is* creativity. Baby steps denote incremental rather than revolutionary progress. To invent an novel interface and application, therefore, requires many baby steps, and as a result the potential for creativity is directly related to the speed with which you can implement the next idea. It is hard to be creative when each prototype revision takes weeks or even months. With the neighborhood viewer, minor revisions could be implemented the same day or overnight and major revisions often could be implemented in only a few days. At the extreme, we would be even more creative if we could prototype revisions as quickly as we brainstorm ideas.

Your flexibility is constrained by the flexibility of your tools. If turn-around is creativity, then rapid development tools provide that creativity. It was essential to our development process to be able to change the interface and the application on short notice to obtain feedback from our users. We used Tcl and Tk because the combination of a scripting language, an interpreter that is active at run-time, and a clean interface to both stand-along programs and compiled C and C++ code gave us the flexibility to quickly modify the interface (while an application is running) and to quickly integrate other applications and code into the application. Our rapid incorporation of wavelet-compressed images is one example of the flexibility of Tcl and Tk. At the same time, Tcl and Tk let us down in certain areas. Until we started to use TclProp, there was no flexible mechanism for connecting many views together. The difficulty in modifying change propagation stunted our ability to expand beyond simple view synchronization to more exciting and innovative interfaces including multi-screen and multi-user interfaces with flexible coupling. Even TclProp was not sufficient by itself, leading us to develop a distributed TclProp. Similarly, Tk provides few hooks to make it possible to monitor changes to widget attributes (only a few attributes have variable connections). We are examining better ways to address this limitation for a future release of TclProp.

Nobody writes advanced programs in a low-level language. As human beings, our thinking is related to our vocabulary. Just as nobody writes a compiler by thinking exclusively about zero and one bits, we could not write the neighborhood viewer until we extended our language to include concepts such as constraints from TclProp, remote procedure calls from Tcl-DP, image handling functions from our wavelet library, and, of course, interface widgets from Tk. Extensions to Tcl are not "extras," they are the core of what makes Tcl worth using. Dynamic loading of extensions and libraries is a good step in the right direction. Still, there is great demand for further support for authors and users of extensions and libraries including: naming and version support, support for user customization, and well-indexed libraries to help people find the code they need.

Don't widen the bottom of the bottle. The bottlenecks in the neighborhood viewer are the image processing, disk, and network access. The wavelet code is all in C (quick experiments showed that Tcl was not fast enough for the computation), but all of the application assembly is in Tcl. Unless it is used for intensive computation, Tcl code is rarely going to be the bottleneck for I/O and computation-intensive applications.

Tcl/Tk is an excellent tool for developing innovative applications – let's keep it that way. We have had great success with Tcl and Tk, and are encouraged by the continuing development of new extensions and libraries that expand our vocabularies and thereby widen our horizons. Experiments with the early alpha version of SpecTcl show promise for another exciting tool that will help speed up our turn-around and increase our creativity. A key concern for the coming years is portability – the ability to distribute a Tk application without sending a new wish, and the ability to easily find and load needed extensions with version control.

Finally, there is one message that is important enough to merit its own section.

Only Dead Systems Don't Change

Each new prototype we developed engendered new interface ideas. Moreover, each prototype brought change to the way that the neuroscientists carry out research. The new interfaces are tools that let neuroscientists learn more about brains, and are not burdens. Indeed, now that browsing is easy we no longer think that the collection of images is large, but rather are looking to fill in the gaps in the "globally sparse, locally dense database."

Depending on how you count, we produced between a few dozen and a hundred prototype versions. Had we expended any effort optimizing the code for any version we would have been wasting our time - the code was destined for an honored burial in the scrap heap. Actually such effort would have been worse than a waste, since we quite naturally would have been loath to discard what we slaved over, and would have resisted change. We did address an evident bottleneck with wavelets, but even there we just modified existing code.

Already, the fifth iteration is completing and we are heading towards a sixth, a seventh, and many many more. With each iteration, we stretch to find the tools we need to support our users. Just as the first four iterations brought us to Tcl/Tk, TclProp, distributed constraints, and wavelets; the next few iterations will lead us to discover or invent abstractions and technologies. And, with each iteration we lift our users just a bit higher, expanding their horizons, and ensuring another iteration of improvement. The story doesn't end here, it is still being written. We wouldn't have it any other way.

Acknowledgements

This research is supported by the National Science Foundation grants IRI-9410470 and IBN-9419233.

We wish to recognize the contributions to this project by our neuroscience and computer science colleagues. Thanks to Chris Honda, Tina Liang, Joe Maguire, Maureen Riedl and Sam Shuster.

References

[1] J. Carlis, J. Riedl, A. Georgopoulos, G. Wilcox, R. Elde, J. H. Pardo, K. Ugurbil, E. Retzel, J. Maguire, B. Miller, M. Claypool, T. Brelje, and C. Honda. A zoomable DBMS for brain structure, function and behavior. In *International Conference on Applications of Databases*, June 1994.

[2] E. Chi, P. Barry, E. Shoop, J. Carlis, E. Retzel, and J. Riedl. Visualization of biological sequence similarity search results. In *IEEE Visualization Conference*, Atlanta, November 1995.

[3] M. Claypool, J. Riedl, G. Wilcox, R. Elde, A. Georgeopolous, J. Pardo, K. Ugurbil, J. Maguire, T. Brelje, and C. Honda. Network

requirements for 3D flying in a zoomable brain database. *IEEE JSAC*, June 1995.

[4] P. Dewan and R. Choudhary. Flexible user interface coupling in a collaborative system. *Proceedings of the ACM CHI '91 Conference*, pages 33–45, 1991.

[5] Sunanda Iyengar and Joseph A. Konstan. Tcl-Prop: A data-propagation formula manager for Tcl and Tk. In *Proceedings of the Tcl/Tk Workshop 95*, page 288. USENIX Assoc; Berkeley, CA, USA, 1995.

[6] Mark Roseman and Saul Greenberg. Building Real-Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.

[7] L. Shoop, E. Chi, J. Carlis, P. Bieganski, J. Riedl, N. Dalton, T. Newman, and E. Retzel. Implementation and testing of an automated EST processing and similarity analysis system. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, 1995.

[8] Brian C. Smith, Lawrence A. Rowe, and Yen Stephen C. Tcl Distributed Programming. *Proceedings of the 1993 Tcl/Tk Workshop*, 1993.