

The Perl Journal

Perl and Air Traffic Control

Richard Hogaboom • 3

Building Custom Tk Widgets in Pure Perl

Ala Qumsieh • 7

Extending Bryar

Simon Cozens • 10

SourceForge Versus CPAN

brian d foy • 13

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

Games, Diversions, & Perl Culture
and *Web, Graphics & Perl/Tk* • 15

Source Code Appendix • 17

LETTER FROM THE EDITOR

Condition Critical

There was a time when interpreted languages like Perl just weren't serious contenders for the really important jobs, the jobs where a failure could endanger human life, or even just open the door to loss of money or property. Interpretation was slow, and "quick-and-dirty-solution" languages like Perl didn't have the trust of project managers. If it was done in Perl, it wasn't a real application, or so the prejudice went.

Today, things are different. The Web has done its part to prove the reliability of Perl, and Moore's Law has eliminated much of the speed gap that kept Perl out of the critical areas. Programmers working on important tasks trust Perl now, and they choose it because it gives them advantages that they don't get from purely compiled languages.

For instance, once you've gotten used to Perl's quick run/test/debug cycle, you never want to wait for a compiler again. This has taught Perl programmers that the true measure of a program's reliability sometimes has more to do with how "fixable" and "tweakable" the program is than a program's raw performance. All code will break someday. A reliable program is one that you don't dread fixing. Perl makes it so easy to make quick changes, you never really dread digging into your source code.

But even in the early days, Perl made some inroads into highly critical areas—like, in this issue, the system Richard Hogaboom describes, that as far back as 1995, used Perl to track the positions of aircraft on the ground at the Atlanta International Airport. For my money, keeping aircraft from colliding is about as critical a job as a programming language can do. It uses some pretty nifty calculus (well, nifty to a math-challenged guy like me), and I can honestly say that it's the only piece of Perl code I've ever seen that needs the speed of light encoded as a constant. I think it proves the point that Perl is up to the job when failure is not an option.

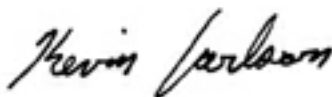
No language guarantees fault-free code. That's up to programmers—and it's no harder to write absolutely dependable code in Perl than it is in C++, Java, or Fortran.

◆◆◆

Back in the summer of 1998, *TPJ* published the article "Just the FAQs: Understand References Today," by Mark-Jason Dominus, that examined the new Perl 5 references feature, which is fundamental to managing complicated, structured data. Eventually, the article became the basis for the perlreftut man page and an important part of Perl's core documentation.

Like you, we think that chopping up established Perl documentation is bad for the language and bad for the community—and it's not the way things ought to be done. That's why the article should remain part of the documentation.

In this spirit, we want to ensure that the article is freely available for your use whenever Perl references are discussed. So if you want to use this article for whatever purposes, feel free to do so without having to ask permission from the CMP lords and lawyers that be, or even us lowly editors. We hope this will keep useful information accessible to Perl users both now and in the future.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

THE PERL JOURNAL (ISSN 1545-7567) is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. **SUBSCRIPTION: \$18.00 for one year.** Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2003 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Song

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, brian d foy, Moshe Bar, Randal Schwartz

CONTRIBUTING EDITORS

Simon Cozens, Dan Sugalski, Eugene Kim, Chris Dent, Matthew Lanier, Kevin O'Malley, Chris Nandor

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEB DEVELOPER

Bryan McCormick

WEBMASTERS

Sean Coady, Joe Lucca, Rusa Vuong

MARKETING / ADVERTISING

PUBLISHER

Timothy Trickett

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403

650-513-4300. <http://www.tpj.com/>

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletta

PRESIDENT, HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenber

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK

COMPUTING MEDIA NETWORK Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

Perl News

Beta System Implemented for CPAN Ratings

Got a favorite obscure CPAN module? Or have you ever wasted hours of time on a module that never delivered on its promises? Now you can let others benefit from your hard-earned experiences. As a “weekend project,” Bjorn Hansen has built a site where CPAN modules can be rated by users: <http://cpanratings.perl.org/>. Each rating must be accompanied by an explanation or review, and only users with accounts at auth.perl.org can submit ratings. Every three hours, the list of distributions and their average ratings is updated into search.cpan.org. The rating system is still officially in beta, but a good number of reviews have already been posted.

Sherzod Ruzmetov made another contribution to the CPAN infrastructure, this one in the form of documentation for new users. His tutorial at <http://author.handalak.com/archives/082003/000340/> details how to prepare new distributions, how to deal with prerequisites, and how to add more files to the distribution.

German Perl Forum Moves

Martin Fabiani posted an announcement to perlmonks.org regarding the German Perl site: “Since the owner of the domain <http://www.perl.de/> doesn’t want to continue the German Perl forum in a noncommercial way, the community decided to found a new German Perl forum at <http://www.perl-community.de/>.” The new site is still under development, but Martin says, “We’ll do our best to keep the good spirit of the former perl.de, and to give a stable and helpful community to German-speaking perl programmers.”

Venues Chosen for 2004 YAPCs

Five cities vied to host next year’s YAPC::NA show, but in the end Delaware, Toronto, Philadelphia, and Washington, D.C. lost out to Buffalo, N.Y. The proposal was created by the Buffalo Perl Mongers via a wiki-style web page, and submitted by Jim Brandt. The University of Buffalo’s Natural Sciences Complex will host the conference; wireless networking is available along with many different sizes of rooms all wired with AV equipment, and a large central hallway for registrations and exhibits. The university’s campus catering service will provide food for the event.

The location for YAPC::EU::2004 has also been chosen—the YAPC Europe Committee decided to give the honor to Belfast, Northern Ireland. In their announcement, the committee gave a

nod to the Copenhagen Perl mongers, who also submitted “an excellent proposal.” Denmark may well host the event in 2005.

RPMPan Comes Online

Linux fan Kevin Pedigo has created a repository of CPAN modules implemented as rpm packages. He writes: “The actual building of the RPM packages is done using `cpanflute2`. A cron job runs every night that downloads the module list from CPAN. This list of modules is then mirrored, and anything new is run through `cpanflute2` to create an RPM. The shell script then goes on to generate the HTML pages, which are then mirrored up to SourceForge.” RPMPan lives at <http://rpmpan.sourceforge.net/>.

Kevin served as technical editor for the book *Linux Toys* (by Christopher Negus and Chuck Wolber: John Wiley & Sons, 2003; ISBN 0764525085), and is president of the Kitsap Peninsula Linux User Group.

Perl \pûrl\, n

The latest edition of the OED includes a new entry for Perl. Enthusiasts of the English language can pick up a copy of the Oxford Dictionary of English (second edition), published on August 21st, and learn that Perl is a noun used in computing to denote “a high-level programming language used especially for applications running on the World Wide Web.” The OED goes on to date the term to the 1980s, originating as a “respelling of PEARL, arbitrarily chosen for its positive connotations.”

Exegesis 6 is Out

If you haven’t yet gotten around to reading Exegesis 6, which Damian Conway released at the very end of July, you’re in for a treat. Beginning with a funny rendition of a hard-boiled detective story (“As soon as she walked through my door, I knew her type: She was an argument waiting to happen”), and ending with an elegant demonstration of why scripters will want to switch to Perl 6, Conway’s explanation of the new subroutine semantics is engaging and enlightening. You can find it at <http://www.perl.com/pub/a/2003/07/29/exegesis6.html>.

TPJ

Perl and Air Traffic Control

Surface surveillance of taxiing aircraft has always been important, and we at Lincoln Laboratory (in Group 42, Air Traffic Control Systems), make its study part of our mission. The public normally thinks of aircraft surveillance as an air-to-air thing—a safeguard against mid-air collisions. But some of the most devastating accidents and losses of life have occurred on the ground between taxiing aircraft and landing or departing aircraft. This article is about how Perl can be useful in improving ground-surveillance systems that guard against such disasters.

Background

Ground surveillance is complicated by several issues that are not present in air surveillance. For most air surveillance, except for low-altitude targets, the sky is the background and is not signal reflective; but for ground surveillance, any radar covering an airport surface must deal with an enormous number of reflective targets, both stationary and moving. Terminal buildings, towers of various types, baggage-handling equipment, and aircraft-towing vehicles are all signal sources that a radar must discriminate against. Seaside airports even have to filter out the ocean itself (waves) or passing ships.

This imposes an enormous burden on any ground-surveillance scheme based on radar. The signal-processing and computational resources required are considerable. No matter how well designed such systems are, they still tend to have several shortcomings: The return signal on a target has no identification; because the radar is both a single source and return receiver, it can miss small targets shadowed by large targets; and the exact velocity of targets is difficult to determine because of the slow speeds. I've seen systems represent targets by moving colored blobs of varying sizes depending on recognition confidence. It's confusing. Sometimes you just have to look out the window.

There is an alternative solution. All commercial aircraft are required to have Mode S transponders. These devices "squitter" signals every half-second that contain the Mode S address of the aircraft (24 hex bits, for example, a9f7c8), which uniquely transforms into the tail number. There are two antennae situated above and below the fuselage that broadcast alternately. In one form of squitter, the long type, the GPS position is also encoded, and receivers located about the airport can forward the information for controller display. The short type of squitter contains only the Mode S address. Most aircraft, at present, only broadcast the short squitter. The scheme devised to locate these aircraft from the short-squitter signal is called "multilateration." In this article, I'll describe an algorithmic implementation of this multilateration scheme in

Perl. This project is from 1995, and reflects some language features from that time. Were I to code it today, I would probably do some things differently.

Theory

The idea is actually quite simple. Several receivers are located at various positions to the side of the airport surface. These receivers are positioned on buildings, the airport surface, or on towers. The ideal configuration is to have all the receivers on the surface of the airport at exactly the same height and in positions that form a lot of equilateral triangles with other receivers. Practical considerations such as the location of terminals and obstacles and the desirability of having the full runway and taxiway surface visible from the receiver antennas make equal height positioning difficult, however.

The target (aircraft) squitter signal is received almost simultaneously by all of the receivers. The "almost" is the important part. Each receiver has to detect the leading edge of the squitter signal to within about 10 nanoseconds (ns). You then compare the absolute arrival times at a pair of receivers to get the time difference of arrival. Since the signal is traveling at the constant speed of light to both receivers (with a little fudge factor for the index of refraction of air), this means a difference of arrival time translates into a path difference that is known.

If you remember covering conic sections in calculus, then you might remember that the loci of points that have a fixed difference distance between two points is a hyperbola (or a hyperboloid) of revolution, because the target may have a height above the plane of the airport surface. You then select another third receiver and find the time difference (distance) in signal arrival of the same squitter and calculate the second hyperboloid.

The intersection of the two hyperboloids will form a hyperbola in a plane perpendicular to the plane of the three receivers. One additional piece of information is needed to fix the target position. For surface targets, this is the known height of the airport surface; and for airborne targets, their altitude. This solution depends on there being one common receiver among the two baselines. Obviously, with a triangle, there are three possible combinations of two baselines. Any of these three may be used. Bertrand T. Fang's paper "Simple Solutions for Hyperbolic and Related Position Fixes," (*IEEE Transactions on Aerospace and Electronic Systems*, Vol. 26, No. 5, Sept. 1990) describes this procedure and also describes solutions with a third piece of information other than height, as well as solutions of time sums rather than differences, which result in ellipsoidal solutions.

Clocks are not perfect, and this type of application requires very accurate clocks. It turns out that maintaining exactly synchronized accuracy is not necessary. All clocks drift to some extent, even atomic clocks. The problem is not to determine the time of day,

Richard Hogaboom works for Solidus Technical Solutions Inc. at MIT Lincoln Laboratory on NASA/FAA research projects. He can be reached at hogaboom@ll.mit.edu.

but only the difference in times of arrival measured in nanoseconds. The solution to this is to provide a reference clock signal from a reference transponder to all the receivers. This transponder is situated in the center of the airport and visible to all the receivers. The reference transponder has its own clock. This clock drifts. It is not important how the reference transponder clock drifts, but how the baseline receiver clocks drift relative to it.

By utilizing the fixed known distances from the reference transponder to each of the receivers (and thus, the known time differences of the propagation of light through air and the times with which each receiver stamps the reference transponder signal), it is possible to do a clock fit and calculate the clock drift of each receiver clock. This drift is then taken into account when doing the time-difference calculations from the actual target signals. The process of position calculation using a triangle of receivers is referred to as “trilateration.”

The several receivers are needed for a variety of reasons:

1. To provide full surface coverage.
2. To provide redundant receivers, because a squitter might be blocked by another target or its own aircraft wing or fuselage.
3. To provide redundancy against multipath signal reflections off the runway or other aircraft that delay and scramble the receiver input.
4. To provide several possible triangle combinations, some of which might provide better geometric characteristics than others.

MLDA (Multilateration Data Analysis)

The choice for the data analysis was Perl. Its impressive capabilities, in the form of built-in functions, made it superior to *nawk* or any shell. Everything about the language seemed well integrated. And I realized that the calculations, even for hundreds of aircraft, were not all that significant for the average workstation. This made the interpreted, quick-turn-around modify/run cycle of Perl attractive.

Atlanta Airport

The data collection part of the project was conducted at Atlanta International Airport in 1995. The receivers, built by Cardion Corp., were called Receiver/Transmitter (RTs) because they could receive squitters and transmit Mode S interrogations as well. There were five RTs and the single reference transponder in the center of the airport surface. Data was taken during normal airport operations and the results sent to us for analysis. All data was in ASCII text.

Because Atlanta is a busy airport, data from the RTs was collected for many aircraft at once. Each time a squitter was received at an RT, its leading edge was time-tagged with an integer 16-bit time with a granularity of 10 ns. A data record with this time, the Mode S aircraft ID, the RT ID, and a confidence indicator was transmitted to a central workstation. All RTs would also continuously receive squitters from the reference transponder. The central workstation would save all this, and the data would be shipped to us.

The mlda Script

The main Perl script is called “mlda” (located in the ml/da/ directory in the package of downloadable source code for this article, available at <http://www.tpj.com/source/>). I won’t try to describe every line of code, but I will go through the script in enough detail to outline its structure and function, and where Perl comes to the fore and where I had to maneuver around it. In this narrative, I am assuming that the reader is following along in the mlda code.

Input Options

The script has only three options: *-s* to skip input file processing if you have a reference transponder file already; *-i ModeSID* if you want to trilaterate on a single target; and *-t c2/c3/i2* if you

want to change the solution method to two- or three-dimensional analytic or two-dimensional iterative. The single argument is the raw RT data. One of the first things I liked about Perl was the easy argument handling; no *getopt()*/*getsubopt()* complexities.

Initialization

BEGIN block processing is next (actually first, since this is done on program *init*, but it’s next in the code). Various tunable parameters are set. These are used to give maximums or minimums for data file processing, such as a minimum number of records needed to do a reasonable solution or a maximum time difference between RT records in order to consider them from the same squitter from the same target. Some physics constants needed for the trilateration algorithm like the speed of light in air are set:

```
# c = 2.997925e8 m/s or 2.997925 m/10ns
$c = 2.997925;

# index of refraction - est. based on air, 20 deg. C, 75% hum.
$ir = 1.000350;

# adjust speed of light by index of refraction
$c = $c/$ir;
```

Next are the RT pairs for which differences in time of arrival are calculated, and the trilateration RT combinations used to do the final position fix. Hashes are used to store the index of the RT combination subscripted by the RT IDs of the pair. Finally, we see the RT locations and the known signal-propagation delays from the reference transponder (based on distance and the speed of light).

Input Data File

The RT data file is input. Perl, of course, makes short work of this. Unfortunately, I do not have any actual old data files from this project. I do, however, have another program, *simpts*, that is designed to generate a fake input file for a single linearly moving target with simulated squitter receptions from each RT with the start point, the direction, the number of points, the point separation, and a little jitter added. The output from *simpts* (in the *simpts* directory in the downloadable source code) is converted into Cardion input file format ready for the *mlda* script.

Each squitter received by an RT generates a record. These have to be merged so that all the data from a single squitter and a single aircraft is together for trilateration analysis. Generally, since the differences in time of arrival are on the order of hundreds to thousands of nanoseconds, and the squitters are broadcast only every half second, then the relevant records will be close together in the input file. Because reception is sometimes scrambled, there are often bit errors in the Mode S addresses. This makes a simple string comparison of two addresses unreliable. We took the approach of allowing two addresses to differ by a tunable parameter (a number of bits) and still be declared equal. Most aircraft have addresses that differ by a lot of bits. We chose 6 bits (out of 24) as a reasonable cutoff. Anything less, and the addresses were considered equal. The merging of low-confidence data into high-confidence data was a goal. The *cntbitd()* subroutine (See Listing 1) does this bit comparison. Perl is often thought of as a text processing language, but I’ve found that it can handle binary files or binary calculations with equal ease.

After confidence merging, the merging or blocking of same-squitter, same-aircraft data is done. First, targets are broken out into entries in a *\$targets/* hash with the Mode S address (six hex digits) as the key. Then all data records have their data *join()*ed into a string, which is pushed onto the array *@\$hex* by the name of the address. I used:

```
@$hex[++#$hex] = join(' ', $idh[0], $swt[0], $rt[0], $scf[0], $sst[0]);
```

Why I didn’t just use a *push()*, I don’t know. If I were writing it today, I probably would. The Perl feature of symbolic named

arrays is one of the most useful syntactic constructs imaginable. Next, loop over *keys(%targets)* and build a new record of all the merged data and write to a file with the address as the name.

Clock Smoothing

The next problem is to do clock smoothing. What's that? Well, no matter how accurate, all the RT clocks will have a slightly different time, and even more important, a slightly different drift. In order to measure differences in times of arrival to the granularity of 10 ns, it's important to take this into account. Other practical considerations arise. At times the reference transponder signal is blocked from an RT, sometimes a target squitter is received by only some of the RTs, and sometimes reception is scrambled so that a squitter is missed (if it's more than 6 bits off).

Another real difficulty stemmed from the specific clocks selected for the RTs: There were jumps in clock readings that were way outside the normal jitter. I refer to these here and in the code as *outliers*. The clocks were oven-controlled crystals. This was a big mistake. These clocks had the necessary accuracy, but they did not drift linearly, or they exhibited quantum jitter within the time periods of the several seconds over which it was necessary to clock fit. The more expensive rubidium clocks are not only more accurate, but they drift very linearly over several minutes, and would have been a much better choice.

I cannot tell exactly how much rubidiums would have improved the clock fits and, thus, the solutions, but I suspect that something of the order of a factor of three to a full order of magnitude in the average root mean square deviation could have been achieved. I had to deal with these outliers somehow. I chose to do two clock fits, one long (over the whole target file), and one short (over the several seconds necessary to eliminate jitter and missed data).

Back to the code. The first thing to do is delete the *%targets* entry for the reference transponder, *a9f7c8*, since its processing will be separate from all other targets; the necessary data is in the file *a9f7c8*. (See line 412 of the *mlda* script.) I then read in *a9f7c8*, which contains the arrival time stamps at each of the RTs for every squitter from the transponder, and calculate the difference of this time and the propagation time of light between the RT and the reference transponder. I then difference this time difference with the propagation difference times to other RTs (all mod 64K since the clocks roll over at 16 bits). Got that? The idea is to see how the clocks on RT pairs are drifting relative to each other from what the difference time of arrival should be from the fixed reference. The relevant line is:

```
$refd{$ref[$rt_cnt], $i, $j} = ( ($tmp[$i*2+1] - $rtd[$i])
    - ($tmp[$j*2+1] - $rtd[$j]) ) % $k64;
```

This saves the difference of differences in a hash indexed by the reference clock time and the RT IDs. If any data is missing, a -1 is entered.

Now comes the C part. I needed to do some kind of smoothing fit on this data, both to get a more accurate measure of clock drift over time and to fill in missing reference transponder data from missed squitters at the RTs. I started out with the idea of a least squares fit, but the outliers made this difficult. I needed something more tolerant of really bad data that would give me some way of obtaining a reasonable first fit that would help eliminate the outlier data and be ready for a second short-term fit.

Numerical Recipes in C (5) (Cambridge University Press, 1993, ISBN: 0521431085) provided a good solution in *medfit()*, or *Least Absolute Deviation Median Fit*. This was much more tolerant of outliers. However, it was not in Perl, so I had to either recode or call it from Perl. I decided to call it. This was where I had to maneuver around Perl the most. I wrote the reference clock time *\$ref[\$rt_cnt]* and the *\$refd{\$ref[\$rt_cnt], \$i, \$j}* to the *clk_cor_raw* file with *prt_refd("clk-cor-raw")*; and called *lad-medfit* with:

```
# do linear clock fit - use Numerical Recipes void medfit() routine
open LSF, "lad-medfit 0" || die "mlda: lad-medfit open failure: $!";
```

```
( @clk_par = <LSF> ) == 10 || die "mlda: lad-medfit completion
failure: $!";
close LSF;
```

This construct runs *lad-medfit* (with a 0 arg which says to long-term fit) and opens a pipe from which the program output can be read. These clock-fit parameters are written to the *lad-par* file for visual analysis. Outlier elimination is then done if any point is too far from the long-term linear estimate. Bad points are written to *lad-out* for visual analysis and are deleted by setting *\$refd{\$ref[\$i], \$l, \$m} = -1*. The new, corrected *\$refd{}* is written to the *clk-cor-out* file. Then the script does a short-term clock fit with the number of points to fit over (*\$st_clk*):

```
# do short term linear clock fit - use Numerical Recipes void medfit()
routine
open LSF, "lad-medfit $st_clk" || die "mlda: lad-medfit open failure:
$!";
@clk_par = <LSF>;
close LSF;
```

I then append the clock-smoothed points to the *lad-par* (again for visual analysis), overwrite *\$refd{}* with the smoothed data, thus eliminating any -1s, and write to the *clk-cor-smt* file. I won't get into the details of *lad-medfit.c* because this is all in C. It's in the *lsf* directory in the source code if you want to see the details.

Solution

Now it's possible to actually do trilateration. We trilaterate for each target file in *keys(%targets)*. These files have the same data format as the reference transponder file, *a9f7c8*, except that they are for ephemeral targets moving into and out of coverage.

Trilaterations are performed on all triangles that have data, and the results are written to **.tls* (trilateration solution) files. Three different types of solution are calculated: a two-dimensional analytical, a three-dimensional analytical, and a two-dimensional iterative solution (1,2,3,4 in "References"). All three are output to the **.tls* files. The routine that calculates the solutions is *ml(\$pt, @rtt)*, where *\$pt* is the master workstation time at which the clock corrections are calculated, and *@rtt* contains the RT time stamps for the squitter in question. The **.tls* files will look like:

```
c3a 544292 0 1 4 1859.990 292.129
c3b 544292 0 1 4 1721.927 494.637
c3 544292 0 3 4 1667.606 571.296
c3 544292 1 3 4 1682.204 556.787
trk 5442921678.509 558.461 3 TK
c3 545342 0 2 3 1669.138 553.378
trk 5453421669.138 553.378 1 TK
c3 545743 0 1 2 Solution not obtainable - abs(R1p-R2p-
R12) > e and abs(R1n-R2n-R12) > e.
c3 545743 0 1 3 Solution not obtainable - abs(R1p-R2p-
R12) > e and abs(R1n-R2n-R12) > e.
c3 545743 0 1 4 Solution does not exist - time differences
inconsistent.
c3 545743 0 2 3 Solution not obtainable - abs(R1p-R2p-
R12) > e and abs(R1n-R2n-R12) > e.
c3 545743 0 2 4 Solution does not exist - time differences
inconsistent.
c3 545743 0 3 4 Solution does not exist - time differences
inconsistent.
```

A normal output with valid solution will show the solution type (c3), the time stamp, the three RTs used in the solution, and the x, y-coordinates (North, South). If the solution fails, a diagnostic is given in place of x,y. If two solutions are admitted (sometimes, the solution is ambiguous) then both are given (c3a and c3b).

For c3 solutions, a tracker is run. A tracker is an algorithm that attempts to eliminate bad data points and smooth over good points. Most trackers take some weighted average over time and attempt to do a reasonable estimate of target movement. This tracker only looks at the solution space of a single squitter solution; that is, all

the trilaterations over any triangle of RTs that had good enough data for a solution, and attempts to eliminate outliers and average over the remaining solutions.

The tracker will drop a track if too much time passes between points, and start a new track. The tracker will eliminate points that show a velocity jump that is too large. The BEGIN block has some tunable tracker parameters. After bad data is eliminated, then an average of the good data is taken and output to the tracker records. If all five RTs receive the squitter and all 10 (5 combination 3) triangle combinations have a good solution, then the average is over 10 points or solutions for the same squitter. Most of the time, this is not true, and tracker solutions average two to five points.

The geometry of the triangles and the target location relative to that triangle is important in determining position confidence. An equilateral triangle with the target in the middle is best; an elongated triangle with the target to the side is less accurate. This has to do with differences in time of arrival being smaller, the ambiguity in position being larger in the odd geometries, and the intersection of two hyperbolas being out on their wings rather than at their inflection points. These *.tls files are then parsed by other simple Perl scripts such as trkx and trix (also in the ml/da/ directory in the downloadable source for this issue) to get data and do plots.

Figure 1 gives you some idea of how all this works. There are three RTs located at x-, y-coordinates (0,0), (1,0), and (1,1). Hyperbola one is a result of difference time measurements between RTs (1,0) and (1,1). Hyperbola two is a result of a different difference time measurement between RTs (0,0) and (1,0). (Actually, their reflection in the lower half plane is also a valid solution, but gnuplot—with which this image was generated—will not plot multiple-valued functions. The intersection of the two curves is the solution. This 2D figure is a simplification of the real 3D case, however, the intersection of hyperboloids of revolution are difficult to visualize, and for a target in the plane, this representation is accurate. Also note the existence of multiple intersection points and thus, multiple solutions.

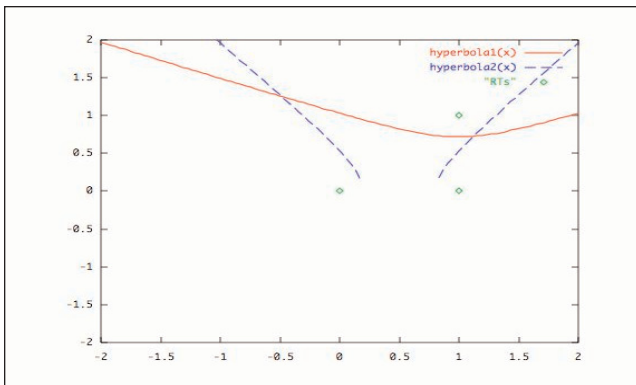


Figure 1: Trilateration solutions for three RTs.

Some reasonable criteria for selecting the single true location are needed. Solutions are often eliminated because they are off the runway surface. If several trilateration solutions with different RT combinations are available, then the common solution would yield the true target location. And finally, the previous target location a second or so before the current solution would be very close. The targets cannot jump suddenly (with very high velocity).

Other Code

There are several other directories to the project. Look at the README file in the project main directory. The most important are the da (Data Analysis), lsf (Least Squares Fit), simpts (SIMulation PoinTS), and anal_sol (ANALytical SOLUTION). A lot of the code in these other directories is in C and represents experimental attempts at better solutions to parts of the problem.

Perl Advantages

I came up with a list of 10 Perl-language features that made it superior to its competitors for this project, at least at the time. These are:

1. Option handling.
2. Dynamic allocation, both initial and expansion of arrays and hashes.
3. Simple I/O.
4. Ease of handling both binary and text files.
5. Symbolic declarations (the `@$hex/++$#$hex/` stuff).
6. Hashes and their multisubscript comma-separated lists.
7. Ease of calling other processes (and getting the data back).
8. Rapid development cycle.
9. Wide variety of useful built-in functions that play well together.
10. Context-sensitive function returns.

This effort was my first significant development with Perl. Before this, I had used Perl for many trivial tasks and got to know the language well enough to feel comfortable using it in a more complex project like this. No doubt the code could be improved with more modern Perl syntax, but the underlying algorithms are sound.

References

1. Fang, Bertrand T. Simple Solutions for Hyperbolic and Related Position Fixes, *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 26, No. 5, Sept. 1990.
2. Ho, K.C., Chan Y.T. Solution and Performance Analysis of Geolocation by TDOA, *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 29, No. 4, Oct. 1993.
3. Friedland, B., Hutton, M.F. *Journal of the Institute of Navigation*, Vol. 20, No. 2, Summer 1973.
4. Boisvert, R. Dr. MIT Lincoln Laboratory, Unpublished Notes and Communications, 1995.
5. William H. Press, et al. *Numerical Recipes in C*, Cambridge University Press, 1993, ISBN: 0521431085.

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

Listing 1

```
sub cntbitd
{
    local($len, $h1, $h2) = @_;
    local($xo, $i, $cnt, @pos);
    local($hex) = "0123456789abcdefABCDEF";

    return -1 if $len > 8 || length($h1) < $len || length($h2) < $len;
    return -2 if substr($h1, -$len, $len) =~ /^(^$hex)/ || substr($h2, -$len,
                                                                    $len) =~
/[^$hex]/;
    $xo = hex($h1) ^ hex($h2);
```

```
@pos = ();
for ( $i=0, $cnt=0 ; $i<$len*4 ; $i++ )
{
    if ( $xo & 1 )
    {
        $cnt++;
        push(@pos, $i);
    }
    $xo >>= 1;
}
unshift(@pos, $cnt);
return @pos;
}
```


Building Custom Tk Widgets in Pure Perl

One of the most useful Perl modules available on CPAN is Nick Ing-Simmons's Tk.pm. Also known as Perl/Tk, this is a port of Tcl/Tk with a Perl object-oriented front-end API. It is used to create graphical user interfaces, and contains a wide variety of widgets that should be sufficient to build the most sophisticated and professional looking GUIs.

Yet, there are times when you would like to add an extra touch to your application to make it stand out among the crowd. While a great number of user-contributed "mega-widgets" exist as modules on CPAN, almost all of them are built by combining native widgets together. While very convenient, this does not add visual uniqueness to an application. Of course, you can always interface directly with the underlying graphic display library via writing a C extension. In fact, a number of such modules exist, but the obvious limitations are the need to understand the low-level Tk API (which requires quite a bit of C experience, due to the API's extensive use of pointers and structures), and the need for users to compile the module before installation.

In this article, I will discuss a way to create visually unique-looking Tk widgets in pure Perl by using the Canvas widget. I do not claim ownership of this idea. In fact, it was suggested to me in an e-mail exchange on the pTk mailing list by Slaven Rezić. Also, I know of at least one Tk module, *Tk::ProgressBar*, that uses this technique, and that comes standard with the Perl/Tk distribution.

The Canvas Widget

The Canvas widget is the most powerful and most versatile of all Tk widgets. In addition to being able to draw primitives like lines, ovals, arcs, polygons, and text, the Canvas can contain embedded images and even other Tk widgets. It also has built-in mechanisms to handle grouping, overlap detection, z-ordering, translation, and scaling. This makes it a powerful tool for the creation of custom-looking widgets. Graham Barr used a Canvas to create his *Tk::ProgressBar* widget, which graphically shows the current value of a certain variable as it advances from a specified minimum to a specified maximum.

One unfortunate property of the Canvas widget is that, due to the inherent window-management mechanism, embedded widgets always obscure other primitives irrespective of the z-order.

Ala works at NVidia Corp. as a physical ASIC designer. He can be reached at aqumsieh@cpan.org.

While this doesn't pose any problems for most applications, it would be nice if embedded widgets could respect their z-order. Being able to draw on *Button* and *Entry* widgets would definitely constitute a deviation from the traditional ways of GUI building, but would open up new pathways for people with fertile imagination.

Upon pointing this out to the Perl/Tk mailing list, Slaven Rezić suggested not using embedded Tk widgets at all, but rather using the Canvas primitives to "draw" the widgets to be embedded. This gave me the idea to create *Tk::FunkyButton* (which is a Perl/Tk widget written in pure Perl and based on the Canvas widget) that defines various nonstandard looking buttons, such as circular and cross shaped. Moreover, we can create special-effects buttons that are not just static. *Tk::FunkyButton* includes two such buttons: a vanishing button, in which the text slowly and repeatedly vanishes and reappears, and a rotating button, in which the text loops around the button (see Figure 1).

Tk::FunkyButton

The first step in creating a Perl/Tk mega widget is to define its base widgets and call the *Construct()* method. I will quickly outline the steps to a mega-widget creation. More detailed information can be found in *Mastering Perl/Tk* by Walsh & Lidie (O'Reilly & Associates, ISBN 1-56592-716-8).

```
package Tk::FunkyButton;

use strict;
use Carp;
use vars qw/$VERSION/;
$VERSION = 0.1;

use Tk::widgets qw/Canvas/;
use base qw/Tk::Derived Tk::Canvas/;

Construct Tk::Widget 'FunkyButton';
1;
```

Because this widget is derived from a Canvas, we use *Tk::Derived* and *Tk::Canvas* as base objects. The *Construct()* method is defined in *Tk/Widget.pm*, and defines a method in the *Tk* namespace with the name *FunkyButton*. This lets us create instances of our *FunkyButtons* like so:


```
$parent->FunkyButton($args);
```

The lonely “1;” is there to satisfy Perl’s requirement that all modules return a true value. Now, we define a lexical hash that maps the possible shapes our *FunkyButtons* can have to the respective subroutines that will do the actual drawing:

```
use vars qw/%shapeToFunc/;

%shapeToFunc = (
    cross    => \&_drawCross,
    circle   => \&_drawCircle,
    rotary   => \&_drawRotary,
    vanishing => \&_drawVanishing,
);
```

Next is to define the *ClassInit()* method:

```
sub ClassInit {
    my $class = shift;

    $class->SUPER::ClassInit(@_);
}
```

This method gets called only once before the first *FunkyButton* instance is created. All it does in this case is to call the *ClassInit()* method defined somewhere in the class hierarchy of our widget, but we can use it to make class-wide adjustments. Since we don’t do anything but call *SUPER::ClassInit()*, we can omit this method, and Perl’s inheritance mechanism will take care of that for us.

When a new *FunkyButton* widget is to be created, the *Populate()* method gets called. This is the main place where your widget springs into existence, and this is where we will inspect the arguments to *FunkyButton* and call the proper function to draw it. It starts like this:

```
sub Populate {
    my ($self, $args) = @_;

    my $shape = delete $args->{-shape} || 'cross';
    unless (exists $shapeToFunc{$shape}) {
        croak "-shape must be one of: ", join(" ", keys %shapeToFunc), " ..";
        return undef;
    }

    my $text = delete $args->{-text}      || '';
    my $cmd  = delete $args->{-command}   || sub {};
    my $relf = delete $args->{-relief}    || 'raised';
    my $bw   = delete $args->{-borderwidth} || 2;
    my $bg;
    if (exists $args->{-bg}) {
        $bg = delete $args->{-bg};
    } elsif (exists $args->{-background}) {
        $bg = delete $args->{-background};
    } else {
        $bg = Tk::NORMAL_BG;
    }

    # now specify canvas-specific options.
    $args->{-background} = $self->parent->cget('-background');
    $args->{-borderwidth} = 0;
    $args->{-relief}      = 'flat';

    $self->SUPER::Populate($args);
```

The two arguments of *Populate* are a reference to the created widget, and a reference to the arguments hash that the user passed.

Since *FunkyButton* is based on a Canvas, our reference is a reference to a Canvas widget that has been blessed into our *Tk::FunkyButton* package. The first thing to do is to extract the information needed from the arguments. As a rule of thumb, try to give default values to each argument and *croak()* if the user supplies something you don’t understand. Once all the info is extracted, we call the *Populate()* method that is defined in the superclass of our widget; in this case, the *Populate()* of *Tk::Canvas* will be called. This is necessary to make sure that all the features of *Tk::Canvas* get defined for us. Be sure to pass the arguments hash reference, after some editing perhaps, to *SUPER::Populate()*.

The next bit of code uses the shape option specified by the user to call the correct function to draw the button with the specified options:

```
$self->{MY_BUTTON} = $shapeToFunc{$shape}->($self, $text,
                                             $relf, $cmd,
                                             $bw, $bg, $spd);
```

The rest of the *Populate()* method deals with informing the window manager with our *FunkyButton*’s desired dimensions and with setting up the bindings for our button so that it responds to mouse events. I will not show this code here—it can be found in *Tk::FunkyButton.pm* and should be straightforward.

Drawing the FunkyButton

The *%shapeToFunc* hash maps the type of button to be drawn to the actual subroutine that draws the button. Currently, there are four types of buttons available: cross-shaped, circular, rotary, and vanishing. All of the subroutines that draw these buttons have the same general steps:

- 1. Compute the size of the button.** This is simply done by drawing the text of the button at (0, 0) and then using the *Canvas::bbox()* method to determine its bounding box. This area is multiplied by a factor, which was determined by trial and error, to get the final size of the button.
- 2. Draw the outline of the button.** To help draw the 3D outline of the button, I defined two subroutines: *_draw3DHorizontalBevel* and *_draw3DVerticalBevel*. Those subroutines are based on Tk’s actual C code that draws the buttons (and are even named similarly), and are used to draw a single horizontal or vertical 3D edge. The arguments to these subroutines are the starting (x, y) location of the edge to be drawn, the edge’s width and height, its relief, whether the edge is “in the shadow,” the miter setting, and any tags that we want to give to that edge. In a normal button, the top and left edges are almost white in color while the right and bottom edges are dark grey. This gives the illusion of a 3D button with some sort of illumination coming from the top left. The shadow option specifies what kind

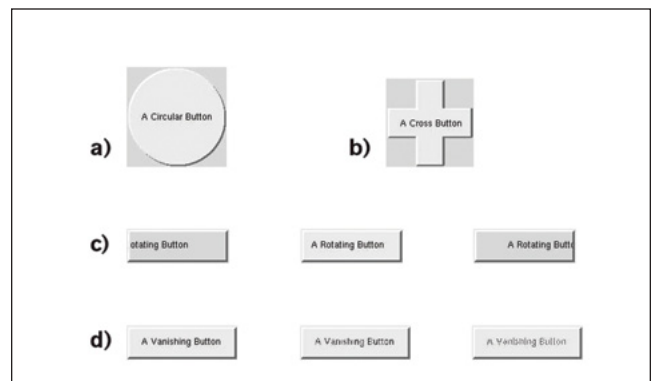


Figure 1: (a) Circle button; (b) Cross button; (c) Rotating button; (d) Vanishing button.

of edge we're drawing, and hence its color. The color is also determined by the relief of the button. The miter setting defines the corner where a light and a dark edge meet is drawn. In case of the cross-shaped button, for example, we have to draw 12 edges, six of which are "in the light" and six "in the shadow."

3. **Draw the background of the button.** Step 3 draws the background of the button with the color specified by the user (or the default system color if nothing is specified). The background is used to respond to the mouse events.
4. **Center the text.** Now that the button is drawn, all we have to do is move the text that we drew in step 1 to the center of the button.

I will not include the code for the `_draw*` methods here, but the code should be self explanatory, especially with the aforementioned four steps in mind. Note that at this point, `Tk::FunkyButton` does not support images, but this can very easily be incorporated into the module.

The text for this button vanishes slowly and reappears repeatedly, as if pulsating

The cross-shaped and circular buttons are the result of the four steps defined earlier. But the vanishing and rotating buttons deserve a closer look since they involve some special effects.

Vanishing Button

As stated earlier, the text for this button vanishes slowly and reappears repeatedly, as if pulsating. The speed of this pulse can be specified using the `-speed` option, which can take a value of *fast*, *normal* (the default), or *slow*. This value is used to look up a corresponding time delay (100 ms for *fast*, 500 ms for *normal*, and 1000 ms for *slow*).

The subroutine to draw a vanishing button first follows the mentioned four steps to draw a regular-looking button with the text centered. In order to achieve the pulsating effect, all we have to do at each time step is to make the text fade a little bit more. Once the text disappears completely, we have to make it reappear slowly. This can be achieved by using stipples. A stipple is a bitmap that is used as a pattern when drawing an object. It acts like a mask to hide certain parts of the object. All we have to do to achieve our fading effect is to define a suitable number of stipples that progressively show less and less of the object. Luckily, Perl/Tk comes with a few bitmaps that prove sufficient for our purposes. Those are:

```
my @stipples = ('', qw(gray75 gray50 gray25 transparent/));
```

Now, we use Tk's `repeat()` mechanism to run a callback at even intervals:

```
my $i = 0;
my $inc = 1;
$self->repeat($delay => sub {
    $i += $inc;
    $inc = -1 if $i == $#stipples;
    $inc = 1 if $i == 0;
```

```
$self->itemconfigure(TEXT => -stipple => $stipples[$i]);
});
```

Initially, we draw the text without a stipple. Then, at each time step, we advance through the `@stipples` array to get the next stipple and use that to draw the text. We have to make sure that once we reach the end of the array, we start going in reverse order. This is done via the `$i` and `$inc` variables, which keep track of where we are in the array, and in what direction we're moving, respectively.

Rotary Button

The text of the rotary button simply scrolls around the button repeatedly. Again, the speed of this rotation is controlled via the `-speed` option. To achieve the desired effect, we use the `Canvas::move()` method to translate the text horizontally to the left. Once the text disappears completely, we simply move it such that its left-most point coincides with the button's right edge and continue with the translation. The code is again very simple:

```
my @orig = $self->coords('TEXT');
$self->repeat($delay => sub {
    $self->move(TEXT => -3, 0);
    my @box = $self->bbox('TEXT');
    if ($box[2] < 0) {
        $self->coords(TEXT => $w + 0.5 * ($box[2] - $box[0]),
            $orig[1]);
    }
});
```

Here, we used the fact that our text is tagged with the `TEXT` string so that we can identify it. To check if the text has completely scrolled off to the left of the button, all we have to do is see whether the right-most x-coordinate (`$box[2]`) of the text is positive or not. If not, we move it just past the right edge of the button.

Getting and Installing the Module

You can grab a copy of `Tk::FunkyButton` from your local CPAN mirror at <http://search.cpan.org/author/aqumsieh/>. The latest version as of the date of writing of this article is 0.01. You can install it using the traditional method:

```
perl Makefile.PL
make
make test
make install
```

Alternatively, since it's all in pure Perl, you can unpack it in any place where Perl will find it.

Conclusion

Tk's Canvas widget is a very powerful widget that can be used to create nontraditional looking widgets. The built-in drawing primitives of `Tk::Canvas`, along with its support for querying, moving, and modifying those primitives make it an ideal canvas (pun intended) for creating unique-looking and dynamic widgets. There is much more potential to be exploited. Any questions regarding the `Tk::FunkyButton` and/or suggestions to improve it are very welcome.

TPJ



Extending Bryar

Simon Cozens

My boss, as I've mentioned before, has some good ideas about software design, and some of them eventually rub off on me. But there's one good idea of his that I'd forgotten about when I was writing Bryar, the blogging software we examined in my last article.

The idea is that if you're writing a class, you should always try writing another—related but functionally different—class at the same time, and then you'll see what concepts can be abstracted out. If your class deals particularly with specifics of a MySQL database, try writing another class to play with a Postgres database. Not only will it obviously help you if you do need to extend your application in the future, it'll show you if all of your concepts are at the right level.

Although I didn't do it at the time, Bryar allows us plenty of opportunities to put this into practice. So, here I present a candid case study of extending Bryar in various directions, together with all the lessons it taught me about putting code in the right place.

Speeding it All up With mod_perl

One of the things that's always been a problem with Bryar is that it's quite slow; it doesn't handle caching, the CGI script isn't persistent so everything has to be recreated from scratch every time a request is made, and so on. With quite a lot of people regularly pulling down XML from my blog, the server is doing much more work than it needs to. So let's make our next extension by converting Bryar to speak mod_perl.

We start by looking at *parse_args*, which receives the path, any arguments, and sometimes the text of a new comment, and turns them into a parsed set of arguments to pass to the *Bryar::Collector*. The *parse_args* subroutine in our CGI version is 44 lines long, a big hint that something is very wrong. Generally, if a subroutine doesn't fit on your screen, it's too big.

This, in our mod_perl version, we extract the URI and the query arguments:

Simon is a freelance programmer and author, whose titles include Beginning Perl (Wrox Press, 2000) and Extending and Embedding Perl (Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumpking. Simon can be reached at simon-cozens.org.

```
sub parse_args {
    my ($self, $bryar) = @_;
    my $r = Apache->request;
    my $pi = $r->path_info;
    my %args = $r->args;
```

And then we find that everything else we need to do will be identical to the CGI version. Oops! Maybe we should put all this into a base class, and while we're at it, we can split up that massive subroutine as well.

Our previously 44-line subroutine now looks like this in the base class:

```
sub parse_args {
    my $self = shift;
    my $bryar = shift;
    my $cgi = new CGI;
    my %params = $self->obtain_params();
    my %args = $self->parse_path($bryar);

    if (my $search = $params{search}) {
        $args{content} = $search if $search =~ /\S(,)/; # Avoid trivial.
    }
    $args{comments} = $params{comments} if $params{comments};
    $self->process_new_comment($bryar, %params) if $args{newcomment};
}
```

This is considerably easier to maintain. It also means that the *Frontend* classes can now concentrate on what they do best, which is dealing solely with the differences between interfaces. As a result, the business end of the *::CGI* class simply becomes:

```
use CGI;
sub obtain_url { url() }
sub obtain_path_info { path_info() }
sub obtain_args { my $cgi = new CGI;
    map { $_ => $cgi->param($_) } $cgi->params
}
sub send_data { my $self = shift; print "\n", @_ }
sub send_header { my ($self, $k, $v) = @_; print "$k: $v\n"; }
```

Again, it's now trivial to work out what this class is doing,

without having to wade through the rest of the code. That makes implementing the `mod_perl` class, and indeed any other front-end classes you want, extremely trivial.

But first, of course, we need to know how `mod_perl` works. `mod_perl` is an extension to Apache that allows much of its internal workings to be driven from Perl. Most `mod_perl`-based applications work by taking responsibility for the content that gets presented to the web client, but you can also use `mod_perl` to write Perl handlers for authentication, authorization, logging, and so on. We're just going to concentrate on content generation.

`mod_perl`'s interface to the programmer comes through the *Apache* object, which represents the request that was made of the server; this is normally called `$r`. It's obtained through the *Apache* `—>request` method, and we can use it to ask the server about the current URL, the path info and so on:

```
sub obtain_url { Apache->request->uri() }
sub obtain_path_info { Apache->request->path_info() }
```

We can also ask it for the CGI parameters, but we need to use the extension module *Apache::Request* to do this, because it more faithfully resembles the CGI.pm interface, and because it handles POST queries as well as GET queries. We use POST queries in Bryar to pass in comments:

```
sub obtain_params {
    my $apr = Apache::Request->new(Apache->request);
    map { $_ => $apr->param($_) } $apr->param ;
}
```

and use the request object to write our headers and data to the client:

```
sub send_data { my $self = shift;
    Apache->request->status(OK);
    Apache->request->print(@_);
}

sub send_header {
    my ($self, $k, $v) = @_; Apache->request->header_out($k, $v)
}
```

And that's basically it, apart from one small thing: Where do we plug this thing in? Apache looks for a subroutine called *handler*, and passes it a *request* object. We'll put our *handler* in the *Frontend::Mod_perl* module too, so that our `mod_perl` handler can be self-contained in the one file:

```
sub handler ($$) {
    my ($class, $r) = @_;
    return DECLINED if $r->filename and -f $r->filename;
    Bryar->go();
}
```

This says that we refuse to handle this request if it's been resolved to a file on disk and that file exists; we do this so that we can have `http://blog.simon-cozens.org/` handled by Bryar, but `http://blog.simon-cozens.org/blog.css`, the stylesheet, handled by Apache normally.

This handler will do what we want, but we can make it a bit more clever by allowing the user to configure Bryar from the Apache configuration file. In my Apache config file, I have:

```
PerlModule Bryar
<Location />
SetHandler perl-script
PerlHandler Bryar::Frontend::Mod_perl
```

```
PerlSetVar BryarDataDir /web/blog
PerlSetVar BryarBaseURL http://blog.simon-cozens.org/
</Location>
```

PerlSetVar sets a variable that we can get at from our Apache request object, if we modify the handler accordingly:

```
Bryar->go(datadir => $r->dir_config('BryarDataDir'),
    baseurl => $r->dir_config('BryarBaseURL'));
```

And now, thanks to good OOP design and abstraction, we have a *handler* in a short, self-contained module that we can drop into a Bryar installation and transform it from a CGI program to an Apache instance.

Generally, if a subroutine doesn't fit on your screen, it's too big

Notice that when we refactored the *frontend* to a base class, the base class asked its subclasses how to do particular things, and they provided specific ways of getting information or performing appropriate tasks. This is a brilliant trick because it completely avoids the need to subclass entire methods. If you apply this appropriately, you can make your classes an absolute joy to subclass. For instance, I wrote *Mail::Thread*, a mail threading library, and wanted it to be useful for each of the various different mail message classes out there, each of which can have different ways of getting headers, and so on. So, when my very own *Email::Simple* library came along, Iain Truskett was able to subclass *Mail::Thread* in very few lines:

```
package Email::Thread;
use base 'Mail::Thread';
sub _get_hdr { my ($class, $msg, $hdr) = @_; $msg->header($hdr) ; }
sub _container_class { "Email::Thread::Container" }

package Email::Thread::Container;
use base 'Mail::Thread::Container';
sub subject { eval { $_[0]->message->header("Subject") } }
```

It's a technique we use often to help design really reusable code.

A New Data Source

However, to get any real speed benefit from our `mod_perl` implementation of Bryar, we will have to write an additional data source that makes use of the fact that we can store document objects in memory as they persist from request to request. But let's not go ahead and do that immediately; we'll apply that trick my boss taught me to implement a completely different data source and check our abstraction layers.

The obvious place from which to get data, if not a filesystem, is a relational database. We'll assume that we've got the following database structure in place:

```
CREATE TABLE posts (
    id mediumint(8) unsigned NOT NULL auto_increment,
    content text,
    title varchar(255),
    epoch timestamp,
    category varchar(255),
    author varchar(20),
    PRIMARY KEY(id)
);
```



```
CREATE TABLE comments (
  id mediumint(8) unsigned NOT NULL auto_increment,
  document mediumint(8),
  content text,
  epoch timestamp,
  url varchar(255),
  author varchar(20),
  PRIMARY KEY(id)
);
```

By far the easiest way to access this database from Perl is for us to create a new *Bryar::Document* subclass. Our new datasource, *Bryar::DataSource::DBI*, will not return *Bryar::Documents* but *Bryar::Document::DBI* objects. This will allow us to wrap the database tables using *Class::DBI* in very few lines of code:

```
package Bryar::Comment::DBI;
use base qw(Class::DBI:mysql Bryar::Comment);
__PACKAGE__->set_db('Main', 'dbi:mysql:bryar');
__PACKAGE__->set_up_table('comments');

package Bryar::Document::DBI;
use base qw(Class::DBI:mysql Bryar::Document);
__PACKAGE__->set_db('Main', 'dbi:mysql:bryar');
__PACKAGE__->set_up_table('posts');
__PACKAGE__->has_many('comments' => 'Bryar::Comment::DBI' => "document");
```

That's all we need. All of the SQL work is done by inheritance from *Class::DBI::mysql*, and all the Bryar side of things is handled by the inheritance from *Bryar::Document* and *Bryar::Comment*.

Except there's a slight nit: *Class::DBI* doesn't really like multiple inheritance and refuses to create a *comments* method if one already exists. So we need to do the inheritance after we've set everything up:

```
package Bryar::Document::DBI;
use base qw(Class::DBI::mysql);
__PACKAGE__->set_db('Main', 'dbi:mysql:bryar');
__PACKAGE__->set_up_table('posts');
__PACKAGE__->has_many('comments' => 'Bryar::Comment::DBI');
use Bryar::Document;
push @Bryar::Document::DBI::ISA, "Bryar::Document";
```

Now we need to write the *Bryar::DataSource* class. As it turns out, there's nothing in *Bryar::DataSource::FlatFile* that can be abstracted out; everything there is flat-file specific. So let's just go ahead and implement the methods we need. Retrieving all documents is easy, thanks to *Class::DBI*:

```
package Bryar::DataSource::DBI;
sub all_documents { Bryar::Document::DBI->retrieve_all() }
```

We have to do a little more work for searching; in fact, *Class::DBI*'s support for searching is less extensive than some of the other database abstraction libraries. However, there's a nice plug-in by Tatsuhiko Miyagawa, called *Class::DBI::AbstractSearch*, which we can use to construct powerful WHERE clauses.

Let's first dispatch the easy case of finding an individual blog post by ID, and then we'll see where *AbstractSearch* gets us:

```
sub search {
  my ($self, $bryar, $params) = @_;
  if ($params{id}) {
    return Bryar::Document::DBI->retrieve($params{id})
  }
}
```

AbstractSearch allows us to construct our WHERE clause in the form of a Perl data structure; if we wanted to search for all blog posts by author *simon* on August 15th, we'd say:

```
{ author => "simon",
  epoch => { "between", [20030815000000, 20030815235959] }
}
```

So we can write our code a little like this:

```
my $condition;
$condition{epoch} = {between => [ _epoch2ts($params{since}),
                                _epoch2ts($params{before}) ] };

if $params{since};
$condition{"lower(content)"} = {like => "%". lc $params{content}."%"}
if $params{content};
```

and we can see if there's a limit:

```
my $limits;
$limit(limit) = $params(limit) if $params(limit);
```

now we can just pass these two hashes to our *Class::DBI*-derived class.

```
Bryar::Document::DBI->search_where($condition, $limit);
```

and this returns a list of document objects!

Next, a quick bit of *Time::Piece* hackery to convert the epoch times we're receiving into SQL-friendly format:

```
sub _epoch2ts { Time::Piece->new(shift)->strftime("%Y%m%d%H%M%S"); }
```

And now we need something to store comments; this is trivial because we can just pass everything to the *new* method of the comment class:

```
sub add_comment {
  my ($self, $bryar, $params) = @_;
  Bryar::Document::Comment->new($params);
}
```

And it's all over bar the documentation: We've added database backing to Bryar in one single drop-in file in around 30 lines of code. This is how it should be.

Genius From Mars

Of course, at this stage, we now have a system that takes documents from a database, renders them with a template toolkit, and spits them out with *mod_perl*. It was at this point that I realized that Bryar may well turn into something more than just weblog software. If you think of the datasource class as data to be modeled, and the rendering layer as a view class, then Bryar acts as the controller in the classical MVC pattern. Combine this with the genericity of Bryar, and you've got something that can be used for displaying catalogues of products, for content and document management, and for all kinds of other purposes.

But let's not go down that road right now; there are three key points we want to take away from our experiences with Bryar for the moment.

The first is that technique of refactoring code by adding another implementation of a class and then abstracting out commonalities, which can lead to cleaner and more extensible code.

The second point, as we've mentioned, is that one way this manifests itself is in having base classes ask subclasses about specific behavior, and implementing the generic behavior in the base class. This leads to wonderfully subclassable code.

The third, less-believable point is that sometimes it pays to listen to your boss.



SourceForge Versus CPAN

brian d foy

CPAN and Sourceforge might appear to be competing services, but they are actually complementary because they focus on different things. I am responsible for about 20 Perl modules available to the public. I use a mix of Sourceforge and CPAN to get the job done. If you are flexible enough to make small adjustments to how you work, you can make the most of both services and spend more time actually working than doing work so you can work. In a nutshell, Sourceforge does well at the general things while, not surprisingly, CPAN excels at Perl-specific things. What follows is a quick guide to when one might be more useful than the other for your purposes.

Where Sourceforge Wins

Sourceforge works for me because I do not want to be a system administrator. I want to focus on writing my software rather than developing tools to support writing my software. Sourceforge may not be perfect, but other people maintain it. It is easy to use for general things I need to do. What are its advantages?

Remote CVS. Sourceforge's biggest advantage to me is its no-hassle Concurrent Versioning System (CVS). CPAN does not have anything like this, so I would have to do a bit of work to let other people commit to any of my repositories if I administered them myself. I could set up local CVS repositories, but then I could not use them if I was at a different computer or I would have to enable anonymous access as well as access for people who want to commit changes. I would rather not be a system administrator, so I let Sourceforge handle that for me. They provide a nice CVS setup, a web interface to the repository, and a one-click process for adding committers. I can still access the repository with the command-line CVS tools.

Collaboration. Sourceforge is ultimately a collaboration tool, and the design of the tools is for projects with more than one member. This adds complexity, but most of it made sense after using Sourceforge for only a couple of days. I do most of the work on

my modules, and I collaborate with Andy Lester who helps out with my *Business::ISBN* module and whom I help out with his *HTML::Lint* module. Working together with him was as simple as adding his Sourceforge user ID to my project—a couple of keystrokes and a click. The Sourceforge system automatically does the necessary things to give him the appropriate access to CVS and other Sourceforge tools. With a little more work, I can set limits or responsibilities for collaborators.

Monitoring. Other Sourceforge members can monitor files that I release on Sourceforge. They will get an automatically generated e-mail that tells them I released a new file and what the changes are.

Releasing nonmodules. Although CPAN is really good for Perl modules, it does not work very well for scripts and other end-user applications. Kurt Starsinic has set up a scripts section, but a lot of people do not know about it, and it has only a handful of scripts. I could easily upload my scripts to CPAN and people could look in my CPAN-user directory to find the scripts, but I think the Sourceforge Files areas handles that better. You can download my cpan script (now included with CPAN-1.63) from Sourceforge, but not CPAN.

Where CPAN Wins

The Comprehensive Perl Archive Network (CPAN) is actually a collection of separate services, rather than a comprehensive and centralized service such as Sourceforge. This decentralization makes it easier to deal with the individual parts. Here's where CPAN has the upper hand:

Canonical source. The CPAN is the canonical source for just about anything Perl; and it would be foolish of me not to put my modules there. People can easily remember search.cpan.org, but not even I can remember the long Sourceforge URLs for my projects. Since CPAN is specifically a resource for Perl and its various front ends, it can focus on the special Perl things, like creating HTML versions of POD files so that people can read the docs before they download the module.

CPAN Search. The CPAN Search web site is at the top of my bookmarks. I visit it several times a day to read module documents or browse a distribution—all without downloading the module. Since it only has to work with CPAN, Graham Barr tailored

brian has been a Perl user since 1994. He is founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and has been a featured speaker at The Perl Conference, Perl University, YAPC, COMDEX, and Builder.com.

it to Perl modules. People can easily find Perl modules, look inside them, go directly to CPAN Testers data, and many more things. Sourceforge has foundries, but they are not very useful to me for more than just grouping similar projects.

CPAN.pm and CPANPLUS. Perl has tools to automate the installation of Perl modules, but my modules need to be on the CPAN so these tools can find them. I firmly believe that the CPAN is one of the reasons Perl is as popular as it is. Python and Java, although fine languages, do not have something as useful or complete.

Automatic testing. The CPAN Testers test almost every module released to CPAN. Some testers do this for every upload automatically, while others do it only for the modules that they need to install. If a module fails, for whatever reason, I get e-mail from the tester that tells me the problem. I do not get this with Sourceforge because its focus is not on Perl modules.

New module announcements. With CPAN, I can find out which modules were uploaded today, or yesterday, or any other day. Several Perl news sources, such as use.perl.org, post this information automatically, and people expect to find it there. Anyone can see these announcements, whereas people need to be registered with Sourceforge to monitor a package. People can also easily discover new modules through CPAN module announcements.

Easy to automate. People actually upload to CPAN through the Perl Authors Upload Server (PAUSE), which is a simple web interface. It is easy to automate with a little web hacking, so I do not have to do anything other than use my “release” command-line utility, which FTPs the appropriate file to the PAUSE anonymous FTP directory, then visits the web site to claim the file for my PAUSE user ID. I can automate various other parts, too. Sourceforge has its own authentication scheme that involves cookies, and although a couple of Perl modules could handle that, I do not need the hassle. PAUSE is a collection of simple forms, whereas Sourceforge has a deeper menu structure.

Bug tracking. Sourceforge and CPAN both have bug-tracking tools, but I find that people who use my modules rarely use them. I put my e-mail address in the documentation of the modules and people write to me directly. I could encourage the use of CPAN’s Request Tracker, but I do not let bugs sit around long enough for that to be a problem. I think that this is more important for users who cannot make the fix themselves (like in big, complicated C++ projects). Most people who report bugs to me seem to be able to send a patch, too, which means they already fixed it locally so they do not need to wait for me to fix it.

Mailing lists. Sourceforge offers mailing lists for projects, and CPAN does not, although the perl.org administrators are very liberal about creating mailing lists. I do not like mailing lists so much, so I tend to stay away from them.

Conclusion

Who wins—Sourceforge or CPAN? Neither. I use the best parts of both. Sourceforge handles the mundane system administration bits and CPAN handles the nifty Perl-specific features.

References

CPAN: <http://www.cpan.org/>
CPAN Search: <http://search.cpan.org/>
CPAN Testers: <http://testers.cpan.org/>
CPAN scripts: <http://www.cpan.org/scripts/index.html>
Sourceforge: <http://www.sourceforge.net/>
brian d foy’s Perl modules: <http://brian-d-foy.sourceforge.net/>
Business-ISBN: <http://perl-isbn.sourceforge.net/>
HTML-Lint: <http://html-lint.sourceforge.net/>

Renew Now & Save!

Plus A Special Offer for Current *TPJ* Subscribers!

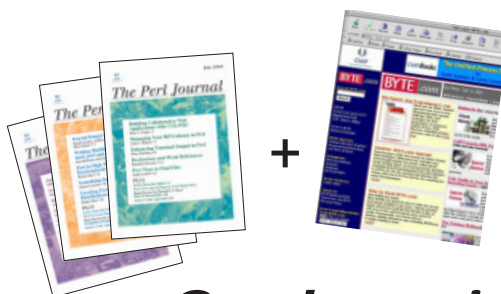
The time for subscription renewal is NOW
—and we have a special offer for **all** current subscribers!

- You can lock in next year’s subscription (and the year beyond that, too!) at the low rate of \$16/year by renewing **now**!
- Have access to the complete *TPJ* archives—Spring 1996 to the present! (Effective September 1, 2003.)
- And by renewing between now and November 1, 2003, you also get one year of access to **BYTE.com** at no extra charge!

That’s a savings of nearly \$20—and you get BYTE columns by Moshe Bar, Jerry Pournelle, Martin Heller, David Em, Andy Patrizio, and Lincoln Spector, plus features on a wide range of technology topics.

Currently, all new subscriptions to *The Perl Journal* are \$18/year. But to show our appreciation for your support in our first year of publication, we’re making this special limited-time offer available to current *TPJ* subscribers who renew between now and November 1.

**Don’t miss out on a single issue or
this special offer! Renew now!**



**= One low price!
One great deal!**

<http://www.tpj.com/renewal/>

The Perl Journal



History Lesson

Jack J. Woehr

I have never had a female intern, nor burgled an opponent's office, nor even traded arms for hostages. Still, I recognize that *The Perl Journal* reviews of a couple of "best of *The Perl Journal*" books may give the appearance of a conflict of interest.

But bear with us. We, the newcomers, who have participated in *TPJ*'s new electronic incarnation can't personally take credit for John Orwant's five marvelous years of paper *TPJ*. Nor do we collect any royalties from the books. While we enjoy basking in the glow of the Perl community's original technical periodical of record, it's a lot to live up to.

Case in point: The recent O'Reilly & Associates releases of *Games, Diversions & Perl Culture: Best of The Perl Journal* and *Web, Graphics & Perl/Tk: Best of the Perl Journal* (following its 2002 "Best of *TPJ*" volume *Computer Science & Perl Programming*).

Games, Diversions & Perl Culture is my favorite of the two. It's sumptuously compounded. The emphasis throughout is Perl as a community of thinkers programming applications that interact with humans in the real world.

Of course, the first word is from Larry Wall, reprinted from the first *TPJ*. In his essay "Wherefore Art, Thou?" (the title is a nerdy pun on "art, n." vs "art, v.pres.2p.sing."), Wall explicitly credits his liberal arts inspiration, planting Perl squarely on the playing field of computer languages for genuine intellectuals, making it, if you will, a modern companion of such ancients as Prolog and Forth.

After Larry's introduction, all 20 *TPJ*'s hard-paper covers follow, reproduced in black and white, and unfortunately, cover art joining the savvy and enthusiasm of Internet Golden Age gurus with a delight in paradox reminiscent of Martin Gardiner and the aesthetic sensibilities of *Mad Magazine*.

That's just the plain fun stuff. The rest is a collection of *TPJ* articles over the years, providing a sampler of the evolution of the Perl practice as demonstrated by some of the more amusing and interesting projects chosen by the authors such as:

Jack J. Woehr is an independent consultant and team mentor practicing in Colorado. His website is <http://www.softwoehr.com>.

Games, Diversions & Perl Culture: Best of The Perl Journal
Edited by Jon Orwant
O'Reilly & Associates, 2003
ISBN: 0-596-00312-9
586 pages

Web, Graphics & Perl/Tk: Best of The Perl Journal
Edited by Jon Orwant
O'Reilly & Associates, 2003
ISBN: 0-596-00311-0
480 pages

Sundials powered by Perl. The Human Genome project. Using Perl to programmatically enhance one's `comp.lang.perl.misc` news-reading experience by filtering flames, spam, and bozos. Controlling X10 home automation with Perl. All this, when you're not busy resolving telescoping images of the cosmos in Perl.

There's Perl-generated Haiku, Perl that is Haiku, and the Obfuscated Perl Contest that presents Perl not as Haiku but as Vögon Poetry.

You'll also find games and game theory such as the Prisoner's Dilemma in Perl, played in team competition, with the most altruistic algorithm scoring dead last in the contest. There are timely political thoughts on secure Internet voting, and homely observations on the fragility of the Perl syntax presented by NORAD programmer Ray F. Piodasoll in his poignant article, "Perl and Nuclear Weapons Don't Mix."

The book has lots of code, lots of theorizing and speculation, and lots of history. All in all, it's a real winner.

Web, Graphics & Perl/Tk: Best of The Perl Journal, is more "technotech," focusing on important linguistic, syntactic, and

algorithmic minutiae, including some important articles from early times in the emergence of popular Perl APIs.

In its section on the Web, this volume presents seminal articles on CGI, HTM, mod_perl, web servers, web spiders, and wireless surfing with WAP and WML.

There's Perl-generated Haiku, Perl that is Haiku, and the Obfuscated Perl Contest that presents Perl not as Haiku but as Vagon Poetry

In the section on graphics, we find Gnuplot, OpenGL, ray tracing, the Gimp, Glade, Gnome, real-time video, and other likely suspects. The Perl/Tk section is focused on using Tk to provide GUI front ends for Perl-coded applications and games. But you already guessed that.

Although *Web, Graphics & Perl/Tk* is less eclectic, inspirational, and soaring than its companion volume, you can't take the fun out of Perl and Perl programmers. If you've ever needed a clickable Beavis, it's found herein. Or if the web content these days (say, for example, the news) seems to you to be a bit surre-

al, you can render it genuinely surreal (at least, for your own viewing) with the HTML Mangler, powered by the travesty algorithm.

Web, Graphics & Perl/Tk is narrower in focus than its companion volume and deeper in penetration. You will read *Games, Diversions & Perl Culture* because you are interested in Perl. You will read *Web, Graphics & Perl/Tk* because you are interested in the particular types of applications referenced in that work. If you possess that interest, *Web, Graphics & Perl/Tk* is a unique resource.

As it has become customary, the web sites for each of these books (see "References") provide tables of contents, author information, sample chapters, example code, and reader reviews.

References

Games, Diversions & Perl Culture: <http://www.oreilly.com/catalog/tjp3/>

Web, Graphics & Perl/Tk: <http://www.oreilly.com/catalog/tjp2/>

TPJ



Renew Now & Save!

Plus A Special Offer for Current *TPJ* Subscribers!

The time for subscription renewal is NOW—and we have a special offer for **all** current subscribers!

- You can lock in next year's subscription (and the year beyond that, too!) at the low rate of \$16/year by renewing **now**!
- Have access to the complete *TPJ* archives—Spring 1996 to the present! (Effective September 1, 2003.)
- And by renewing between now and November 1, 2003, you also get one year of access to **BYTE.com** at no extra charge!

That's a savings of nearly \$20—and you get **BYTE** columns by Moshe Bar, Jerry Pournelle, Martin Heller, David Em, Andy Patrizio, and Lincoln Spector, plus features on a wide range of technology topics.



+



= **One low price!
One great deal!**

<http://www.tpj.com/renewal/>

The Perl Journal

Source Code Appendix

Richard Hogaboom “Perl and Air Traffic Control”

Listing 1

```
sub cntbitd
{
    local($len, $h1, $h2) = @_;
    local($xo, $i, $cnt, @pos);
    local($hex) = "0123456789abcdefABCDEF";

    return -1 if $len > 8 || length($h1) < $len || length($h2) < $len;
    return -2 if substr($h1, -$len, $len) =~ /[^\$hex]/ || substr($h2, -$len,
                                                                    $len) =~ /[^\$hex]/;

    $xo = hex($h1) ^ hex($h2);

    @pos = ();
    for ( $i=0,$cnt=0 ; $i<$len*4 ; $i++ )
    {
        if ( $xo & 1 )
        {
            $cnt++;
            push(@pos, $i);
        }
        $xo >>= 1;
    }
    unshift(@pos, $cnt);
    return @pos;
}
```

TPJ