PostScript-as-Language Runtime Speedup Tricks

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

The first few cuts of our Swings & Tilts utility in GuruGram #15 ran kinda slow, and it did take me several more passes to get it up to reasonable speed. What I thought I'd do here is review some of the methods you can use to optimize your PostScript-as-Language runtime speeds.

We'll first note that too much speed optimization may lead to code that is longer, more obtuse, harder to maintain, and less tutorial. So a balance between speed and usability is often best. PostScript has gotten amazingly fast over the years. Consistent with what is possible within the limits of a batch-oriented interpreted higher level language. Here are some items to consider when optimizing your code for fastest runtime speed...

Write your own code — Most commercial packages that generate PostScript code write sequences that are ridiculously long and execute way too slowly. Your own "bare metal" hand coding can eliminate anything unnecessary and should run ten to a hundred times faster. My Gonzo Utilities are one route towards fast and efficient high performance routines.

Get your code working first — The first cut of any program can always have its speed improved, so don't worry too much about optimizing speed early in the game, Use a two-step process where you first get your code to do exactly what you ask of it. Then go back through and clean up any speed issues.

Use a fast machine and the latest version Distiller — PostScript-as-Language code will obviously execute faster on a new 4 GigaHertz computer than on an older 50 MegaHertz one. Especially if the latter does not include a decent math coprocessor. The latest Acrobat Distiller versions have gotten stunningly fast.

Use print to disk — The standard way of capturing and evaluating slow PostScript code is to do a print to disk and then inspect your results as an ordinary ASCII textfile. The code can then be analyzed for any excesses, redundancies, or speed limitations. But note that certain newer Acrobat features such as transparency have no direct PostScript equivalent and thus lead to large and slow code.

Don't Sweat mul or div — At one time, more complex math functions were quite slow, but these days with a modern math coprocessor, the PS mul multiply or div divide operations usually run just as fast as ordinary addition or subtraction. Even trig functions such as a cos cosine calculation will only take four or so times longer. Allow a microsecond or two per trig function.

Bound and Determined — Your first and foremost speedup trick is to use the PostScript bind operator. Simply place a bind before each major def. This tightly links variables to their values and eliminates a lookup step. Binding typically speeds you up by fifteen to twenty percent. But may not always be appropriate.

Measure program speed — The PostScript usertime operator measures and records one millisecond ticks. Longer times may be evaluated by repeated use. Here are some handy procs from my gonzo utilities...

```
/starttimer {usertime /mytimenow exch def} def
/stoptimer {usertime mytimenow sub /mytime exch mytime
    add def} def
/resettimer {/mytime 0 def} def
/reporttimer {mytime 1000 div (\rElapsed time: ) print 20
    string cvs print ( seconds.\r) print flush} def

/stopwatchon {resettimer starttimer} def
/stopwatchoff {stoptimer reporttimer} def
```

Normally, you just put a stopwatchon at the beginning of what you want to time and a self-reporting stopwatchoff at the end.

Big lumps first — Your first speedup goal is to measure how much time you are spending in each activity. Normally, the innermost code loops will dominate since they tend to execute much more often. Obviously, your first efforts at speedup are best spent in those areas that are using up the most time.

Try null differencing — Sometimes it is tricky to measure a complex inner loop of major code. A sneaky way around this is to measure your total program time, then comment out the inner loop and measure again. The difference between the two should be your inner loop time. Repeated differencing can then get your total time budget for typical program operation.

Measure proc speeds — Individual PostScript commands or small procs will execute much faster than a millisecond and thus give you a usertime value of zero. Instead, measure the time of one million procs at once...

```
stopwatchon
1000000 { your_proc_here } repeat
stopwatchoff
```

Select store instead of def — The PostScript def command always assigns new vm resources, where store writes over previously assigned values. Stores thus can be far more compact and efficient.

Stuff the stack — PostScript stack operations are self-addressing and the fastest available. Creative use of dup, copy, index, exch, and roll are often the secret to tight code. The more you can do with the stack, and the deeper you can load it, the faster your results will often be.

But watch for excessive exch — The exch command is certainly among the most versatile and fastest in PostScript. But too many of them in the innermost tight portions of your code strongly suggest that you need to rearrange things or earlier preload other items on your stack. Ferinstance, a /cx exch def can sometimes get its exch eliminated by a previous /cx stack entry.

Reuse strings — Predefine a workstring or two and reuse them over and over again, rather than creating a new string for each and every use instance. But be sure to apply the new string content immediately, or it might change rather unexpectedly later in your routine. Note that strings are not protected by the usual saves and restores.

Use table lookups — A key rule is to never calculate anything you already know the answer to. Table lookups are one of the greatest and least appreciated routes towards fast code. With a table lookup, you simply go get the answer, rather than spending a lot of time calculating it. Here is an Elegantly Simple example where a PostScript lookup table fakes the case command of other languages...

```
{{proc0} {proc1} {proc2} {defaultexit}} exch get exec
```

Entering with a number 0, 1, 2, or 3 does proc0, proc1, proc2, or defaultexit without having to test for each selected value. As a fancier example, very complex bicubic basis functions are needed for pixel interpolation. Instead of calculating them over and over again, you simply look up the results...

```
/b1 [1.000 0.999 0.998 0.992 0.979 0.954 0.916 0.863 0.795
0.715 0.625 0.529 0.432 0.338 0.252 0.176 0.113 0.064 0.028
0.007 0.000 ] store
```

You can inspect SWINGT01.PSL as a detailed use example. This code is discussed in depth at GuruGram #15.

Table lookups can either be downloaded or calculated early in the program. On-the-fly table building makes sense so long as the build time is much less than the use time saved. While table sizes of a few hundred bytes are the norm, very large lookup tables can sometimes be used to advantage. Tables are often best as one-dimensional arrays.

Minimize disk reads and writes — The PostScript disk access operators quite useful but rather slow. If you are going to need thousands or millions of disk reads or writes, consider using the writestring and readstring operators to create long buffer strings. Use of put and get on those buffers can be much faster. Especially when processing, say, an entire scan line of pixels.

Watch those "==" and flush instructions — It is certainly a good idea to report lots of results often to your log files. But the flush operator can be exceptionally slow, and it is needed by == or anytime you want an immediate text output. Here's an outer loop "still busy" status reporter that adds little overhead time...

```
dup 24 mod 0 eq {(.) print flush} if  % for count on stack
```

Use Conditionals with care — The PostScript if and ifelse statements execute rather slowly, so you often will want to avoid any and all unnecessary testing. Sometimes and or or Booleans can be successfully used to reduce multiple tests to a single one. Other times, rethinking of an algorithm can let one test do the work of two. Or place one inside the other.

Calculate changes only — If you are making very complex calculations and only a small part of the calculation changes each time, consider saving old results and recalculating only the changes. This dramatically sped up my Magic Sinewave research. Sometimes derivatives or differentials can also be used to quickly find a neighboring value. This is an advanced concept I'd be glad to help you with per our Consulting Services.

Be aware of Garbage Collection — Earlier PostScipt code had a disconcerting habit of stopping every now and then to do vm cleanup housekeeping. Which could add mysterious and erratic long delays to your processing time. One obvious precaution is to shut down other host programs when running Distiller. A second defense is to avoid excess buildup of abandoned strings, defs, variables, saves/restores, and whatever in your code development. And a third is to be aware of the vmreclaim command and its use. As detailed in the PostScript Level II Reference Manual and the PostScript Level III Reference Manual.

Some random tips — Minimize the total number of variables. Do as much as you can directly on the stack. Don't sweat exact name lengths or definition sequences as these do not seem to consistently save you a lot of speed. It's usually better to keep names informative rather than ultra short. Use "linear" coding instead of nesting "subroutine" proc after proc after proc. But it is probably a good idea to keep your main code loop clean, calling only a few key detail procs as needed. Consider glyph, font, or Xobject tricks where you need to reuse a complex but small image. Debugging is often enormously helped by purposely introducing a zorch "stop here" error. Test and measure continuously. For "obvious" speedup tricks may in fact end up slower than you'd first expect.

Think outside the box — If things are still too slow, perhaps a radical new algorithm would be better. Maybe parallel processing on several machines at once (this works like a champ on my Magic Sinewaves.) Or, in the case of those Uh-compared to what? situations, running overnight, or on a secondary machine, or during lunch can be effective.

In several instances, I've successfully used easy, powerful, and convenient interpreted PostScript-as-Language for all of my early development work and later switched to another language for the final out-the-door product. I've even used PostScript to automatically write JavaScript programs! Or even to create a Custom Robotic Languages.

For Further Help

Additional background along with related utilities and tutorials appears on our GuruGram, PostScript, Acrobat, and Fonts & Bitmaps library pages.

Consulting assistance on any and all of these and related topics can be found at http://www.tinaja.com/info01.asp. As can our speed analysis services.

Additional GuruGrams await your ongoing support as a Synergetics Partner.