Writing commercial grade multiplatform end-user applications with Tcl/Tk and PowerTcl

Bastien Chevreux[*], Christoph Göthe, Sebastian Liepe

MWG Biotech AG
Anzinger Str. 7, 85560 Ebersberg, Germany

Abstract

This paper is a kind of front line report on the ups and downs while using Tcl/Tk and PowerTcl with scripts and applications that belong to one of the most demanding domains that exist nowadays: bioinformatics.

The paper shows where the multiplatform compatibility (Unix, Windows and MacOS) of Tcl/Tk helps development of applications in estimated 25% to 30% of other languages like Java or C/C++, where running applications cross-plattform comes as a bonus.

But it also depicts some of the limits set to it. A large part of the paper is dedicated to an in depth analysis of a seemingly simple algorithmical problem that shows how algorithms must be rethought when programmers are more used to low-level languages.

1 A quick introduction to MWG Biotech AG

MWG Biotech AG is a company dedicated to customer services and research in the genomic area. This includes

- a large scale DNA synthesis facility (modified and unmodified nucleic accids)
- custom sequencing laboratories
- a genomic diagnosis division designing and producing oligonucleotide based microarrays for different organisms as well as doing the same for customer specified wishes
- automated biosystems for complete automation of applications including high throughput sequencing or the production of microarrays
- a bioinformatics department to provide the required computer science needed for all the tasks above.

In the bioinformatics department is a group of 11 people where quality of work, speed and cost efficiency are the most treasured properties, we therefore have a very pragmatic view and use the tools suited best for a task. On the hardware side, there are big Sun servers for computation and databases, specialized hardware (Paracel GeneMatcher and Paracel BlastMachine) for very specific tasks and a few Windows PCs and Macs for cross-plattform development. On the software side there is a coexistence of C/C++, Delphi, shell scripts, Perl and – of course – Tcl. Six people are currently working almost exclusively with Tcl or TclX for inhouse scripting and customer software on both Unix and Windows platforms, with a push to keep Mac version up to date.

1.1 The data we're working with

Two business segments of MWG, namely genomic sequencing and oligo design, produce and/or use genomic data extensively. This section just gives a short insight with what kind of data a bioinformatics department have to deal with ... amongst other things, that is.

[*] To whom correspondance should be adressed: bach@mwgdna.com

1.1.1 Genomic sequencing data

Todays genome sequencing projects produce enormous quantities of data each day. Gel or capillary electrophoresis used can determine only about a maximum of 1,000 to 1,500 bases, the high quality stretch with low error probabilities for the called bases often being around the first 400 to 500 bases. Each base needs about 270 bytes of raw data, each file will take (unpacked) about 260kb (around 50kb packed).
Because of errors happening during the data collection even in the higher quality stretches, every base of a genome being analyzed should be *covered* present multiple times in different electrophoresis data, todays accepted *coverage* is approximately 6 times, but can go up to 12 or more.
Current sequencing strategies for a contiguous DNA sequence or even whole genome approaches therefore basically boil down to fragment the given contig in thousands of overlapping fragments, analyze these by electrophoresis and subsequently assemble the subclones back together in one contig. Taking a normal sized bacteria of about 2 megabases as example, one would have a minimum of 2M / 500 * 6 = 24000 fragments files with about 1.2 gb of data. This is by the way only the raw data, sometimes up to 10 files with related data are generated from this raw data file.

1.1.2 Genomic databases: data for oligo design

Genomic projects like the above and smaller genomic data snippets are collected worldwide in so called sequence databases. The term database being used a bit lightly, because in many cases these databases essentially consist of flat files. As an examples, the Genbank database contains about 17M sequences with around 20G bases (numbers taken as of beginning of June 2002). As additional textual information is also stored, the resulting database is around 80G big (packed: 10G).
An oligo (a short term for oligonucleotide) is a small stretch of DNA or RNA, mostly between 5 and 100 bases long. Designing oligos for genes is the process to search the database for hopefully unique sequence stretches according to a give set of criteria that match only to this particular gene. Apart from massive parallel computing power this needs sophisticated programs that extract and mangle sequences from all available data sources.

2 FeatureViewer: driving "wish" to the limits

As already explained, sequencing genomes leads to a plethora of data and information snippets. These have to be visualized e.g. when doing explorative research on a completely sequenced genome. Figures 1 and 2 show an example on how this could look like in Tcl.
One important thing to keep in mind is that some of the genomes we would like to visualize have up to 100 million bases. Taking a small conservative font with eight pixel per character, this leads to canvas sizes with up to 800 million pixel in width. Although this may sound unbelievable, Tcl has no problem in creating such canvases, neither on Unix nor on Windows.
With one notable exception: one cannot create easily several of these canvases stacked together (a multi-row canvas) as we need them when we want to blend in or out different rows of information.
Some short examples of this effect are available on the Tcl'ers Wiki: http://mini.net/cgi-bin/nph-wikit/3470.html
Even small genomes have tens of thousands of annotations, i.e., comments made to the sequence by algorithms or by hand. Each of these annotations is represented by one or several objects on a canvas, leading to canvases sometimes containing a hundredthousend objects or more.

3 OligoManager: fast path development in Tcl

MWG not only designs oligonucleotides but also produces them. Customers can order oligos via email, web ordering form or now with a special oligo management and ordering program which was developed in Tcl in about one man year. Tcl was chosen because programs run across three platforms with just a little overhead in porting programs.
OligoManager uses these packages:

- BWidget ToolKit, part of tcllib (http://sourceforge.net/projects/tcllib/)
- EazySMTP package
- Tablelist package (http://www.nemethi.de/)
- Wcb package (http://www.nemethi.de/)
- HTMLDisplay package

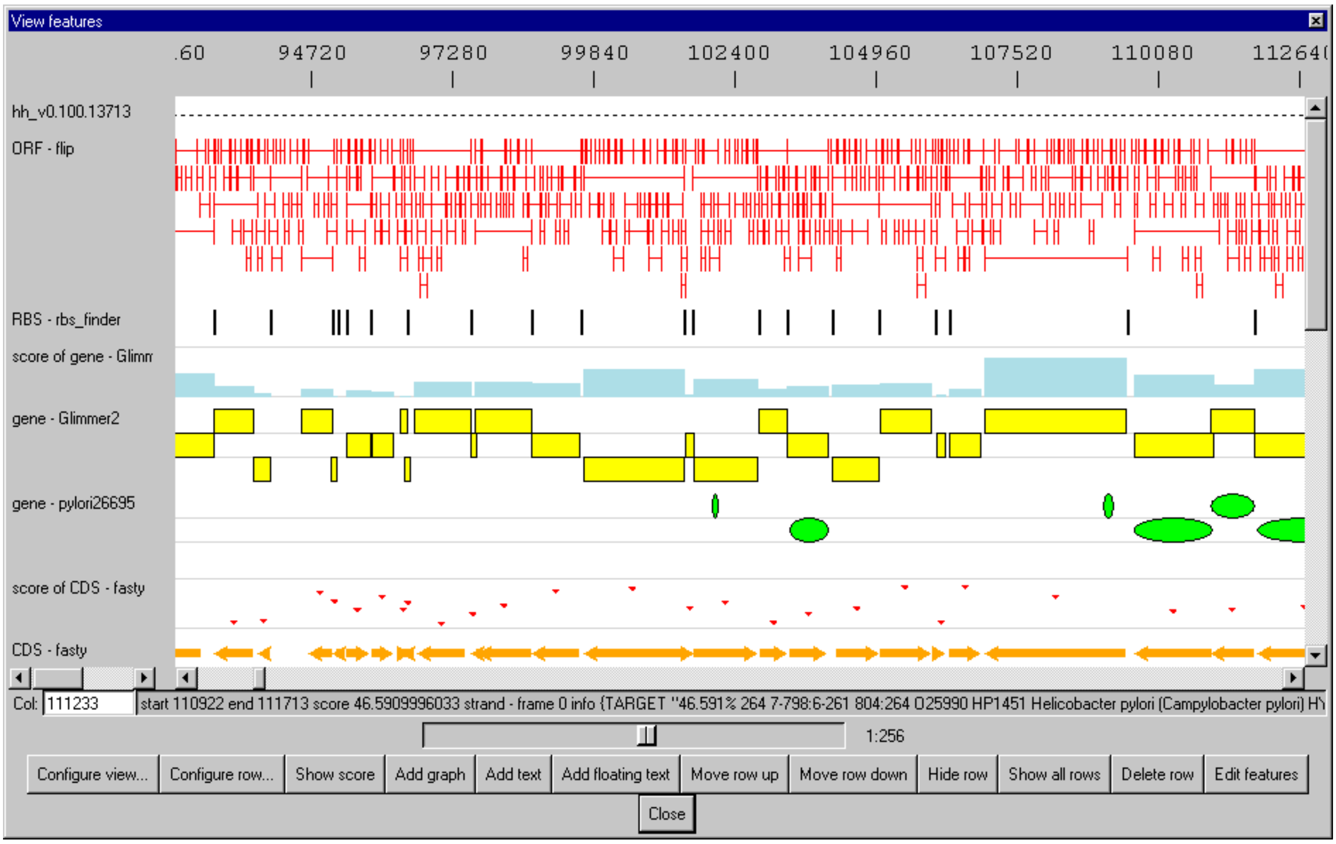

Figure 1: An example on how to visualize different areas on a genome. Every 'row' is a canvas on its own that can be shown or unmapped, sorted upward or downwards.

- TclLib (http://sourceforge.net/projects/tcllib/)
- winutils (http://sf.net/projects/tomasoft/)
- PowerTcl compiler (http://www.compiler-factory.com/)

Figure 3 shows a screen shot of version running on Linux, Figure 5 almost the same version running on Apple, albeit with some irregularities because some necessary adjustments were not made. The talk itself will probably use the Windows version for demonstration.

3.1 Small hacks explained: round corners for BWidget titleframes

Todays 'modern' GUI tend to prefer smooth objects and round corners for screen widgets. A small hack was written to see whether this might be possible for some simple things in Tcl and the BWidget Titleframe was chosen because it seemed easy enough to incorporate into, and it was.

So how does one get round corners when all Tcl can do is drawing 90 degree corners? Well, fake them. The basic idea is simple: let BWidget draw the titleframe and put other widgets – in this case images containing the round edges – over the corners. The images themselves are 4 by 4 pixel where each pixel is either assigned a foreground color (the color of the frame) or a background color (color of the background 'below' the frame.

Three pixel also get an intermediate (anti-aliasing) color to make the corners look more round. The formula to calculate this anti-aliasing color is a gamma corrected interpolation, because calculating the average of both values would give a color which

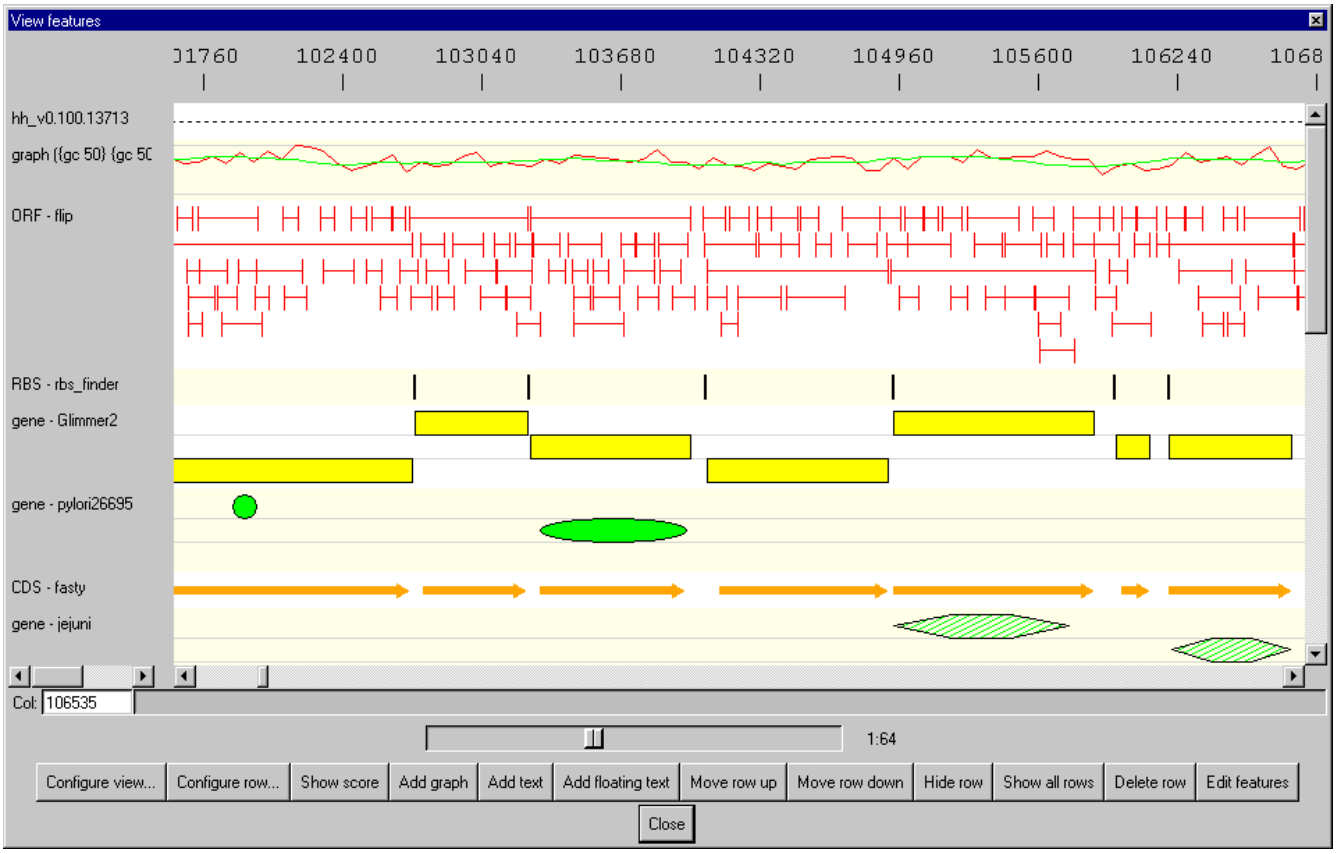

Figure 2: Another exemplary view on the same small genome.

looks too dark. The formula for gamma corrected interpolation is

$$(0.5 * [(R_1^{2.5}, G_1^{2.5}, B_1^{2.5}) + (R_2^{2.5}, G_2^{2.5}, B_2^{2.5})])^{0.4} \tag{1}$$

where R_x, G_x and B_x are the values for red, green and blue respectively, for each color. The example for the color green and magenta would give

$$(0.5 * [(0^{2.5}, 255^{2.5}, 0^{2.5}) + (255^{2.5}, 0^{2.5}, 255^{2.5})])^{0.4} = (193, 193, 193) \tag{2}$$

which, incidentally, is a light grey.

3.2 Nifty tips explained: using pages or notebooks for faster GUI responses

One of the main problems when using complex GUIs created with BWidget is speed. Simply building a GUI can take noticeable time and this is one of the things users hate and they'll blame a program for it.
One nice trick in is to build most of the GUIs one needs and then to map and unmap the windows when needed. In some cases, using the BWidget Notebook or PageManager not only leads to a clean user interface, but to quick response times, too. Notebook and Pagemanager are provide basically the same functionality over window 'overlays', except that the notebook comes with a user ready interface (the tabs) while the pagemanager can be addressed by any means (menus, buttons, lists, etc.).

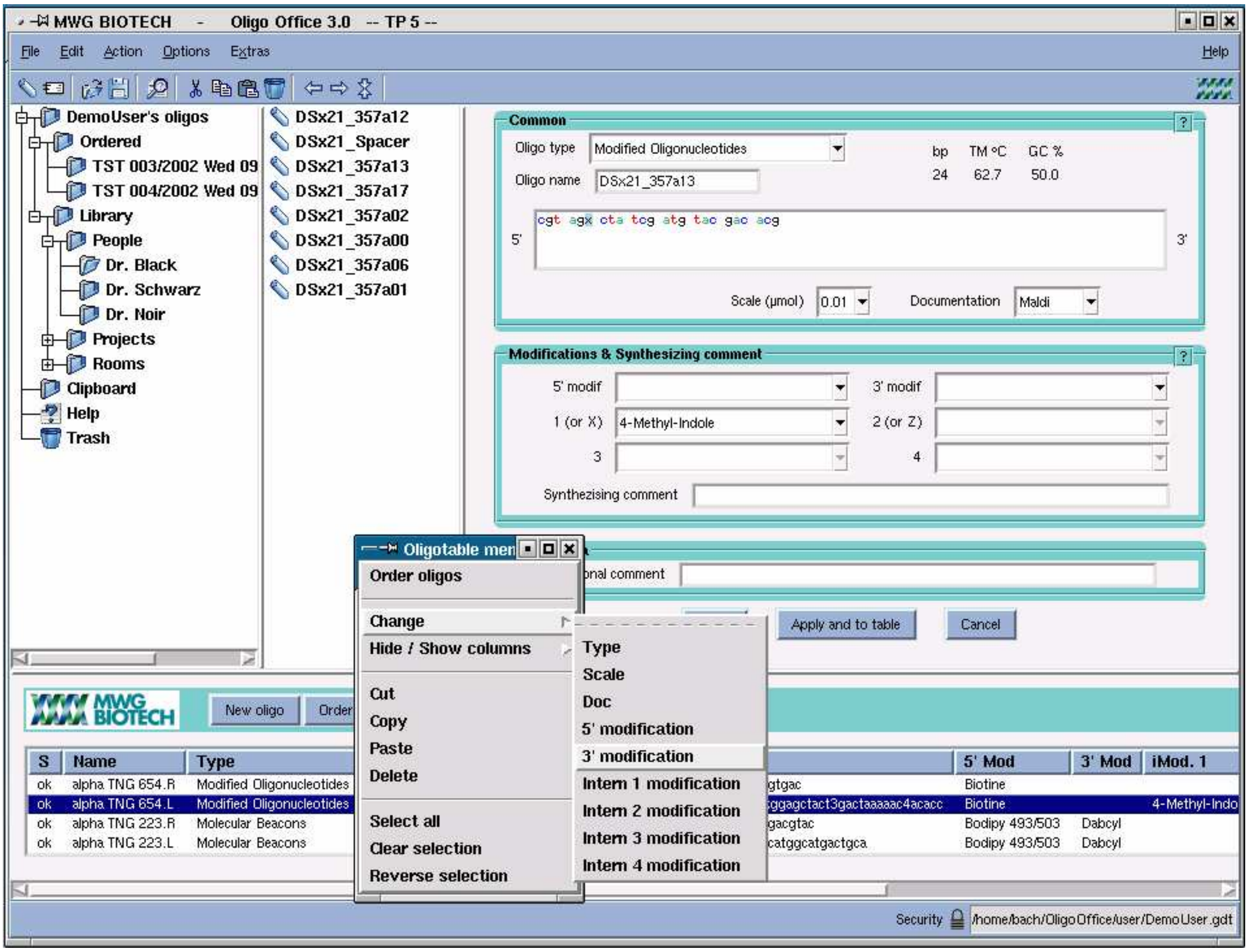

Figure 3: The OligoManager is a customer convenience application developed in one man year and that integrates into the existing ECommerce environment present at MWG.

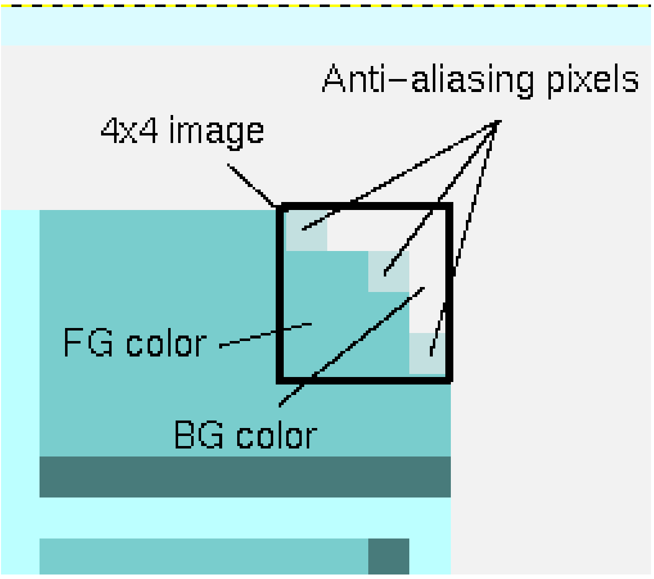

Figure 4: Faking round corners with overlay images.

3.3 Nifty tips explained: using named fonts

It's very easy to write GUIs like this:

```
button .b -text Something -font {helvetica 10}
```

While this might be good for small hacks or trying out something, it isn't very flexible if you'd like to change the look of an application somewhen during the development process as finding all occurrences of this string in a source code can get tedious. The first way in which this could be improved is to put the font definitions in variables and use these. E.g.

```
set butFont {helvetica 10}
...
button .b -text Something -font $butFont
set a $bla
```

This can also be used to change the look at runtime ... sort of. Because all the widgets already created will not change their look when the variable changes. E.g.:

```
set butFont {helvetica 10}
...
button .b -text Something -font $butFont
...
set butFont {fixed 20}
...
button .b2 -text Something -font $butFont
```

Although button .b2 will then have the new font, button .b will not. But this can be addressed, too, by using named fonts. All widgets, even those already existing, immediately adopt the new font when it is changed, even when they are already draw. E.g.:

```
font create butFont -family helvetica -size 10
...
button .b -text Something -font butFont
...
font configure butFont -family fixed -size 20
```

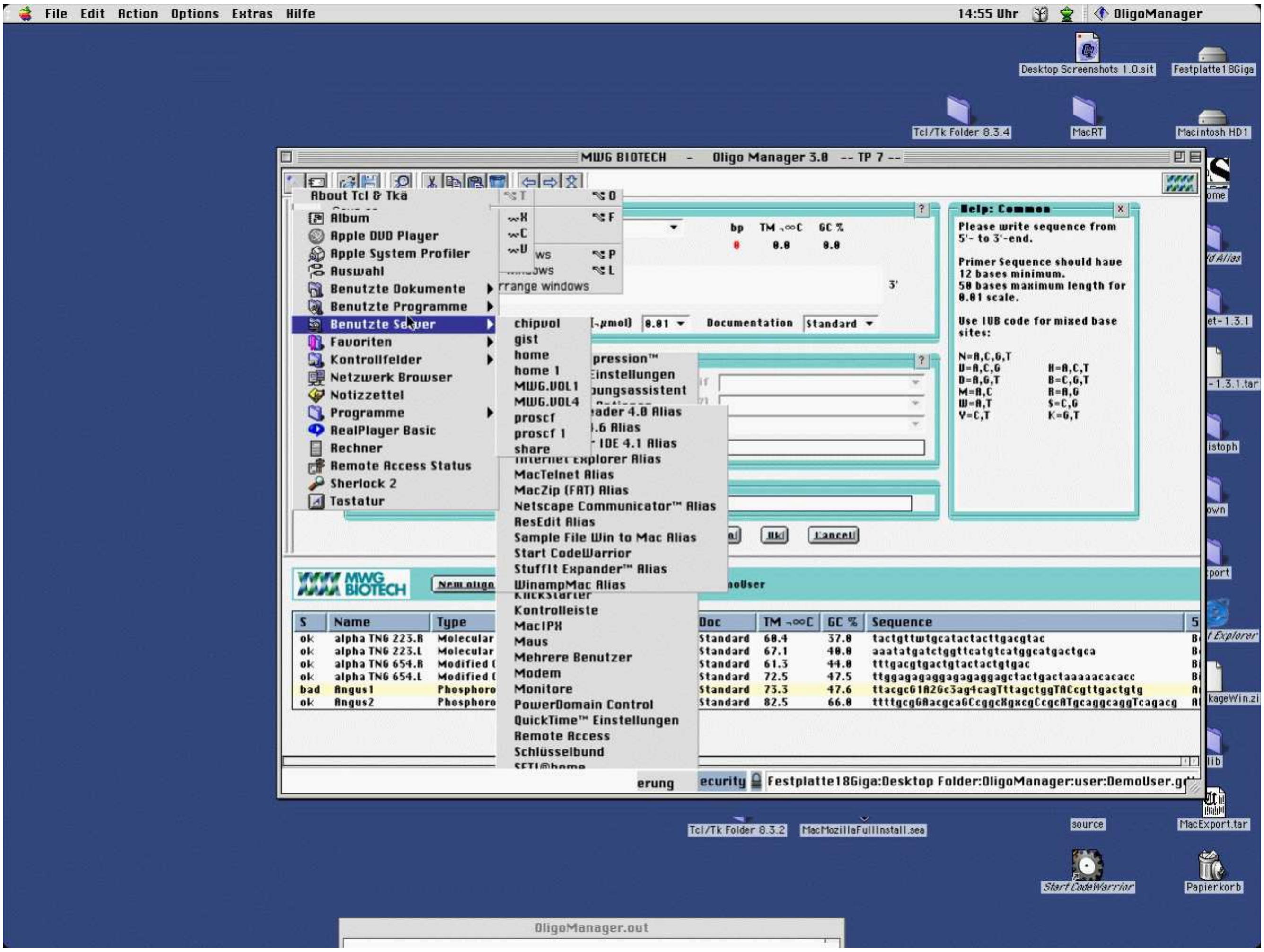

Figure 5: Funny 'feature': the Apple system menu is gridded *within* the application and not in the top menu bar.

```
...
button .b2 -text Something -font butFont
```

Now button .b will immediately change its shape once the font configure command runs, thus showing a consistent look everywhere.

3.4 Expected and unexpected Macintosh woes

In the process of porting different applications to Apple Mac, we discovered a bunch of undocumented funny 'features' and show here the most graphical one for the menus. In some cases, the menus will be shown not in the top Apple menu bar, but instead relatively to the application just as if it tried to behave like Unix or Windows applications, see figure 5. Needless to say that this causes quite some confusion both with the user (who does not expect that behavior) and the programmer ... who also does not expect that behavior and has no idea how to stop it.
Another thing to keep in mind on Mac (OS 8 and 9) is that third party libraries need special attention when trying to use them. Here's just a small list of things to keep in mind:

- In case of Tcl packages, authors often forget the existence of basic crossplattform commands, e.g. the `file` command.

While Tcl for Windows will happily transform pathnames with a slash ("/") to pathnames with a backslash ("
"), this does not happen on Mac.

- Small differences in 'Wish' behavior make enhanced functionality, like e.g. bubble-help windows, difficult to port.
- Porting C libraries is notoriously difficult if the authors didn't take precautions to make things easy. One good example is the TLS library, itself being not to complex but which in turn needs the OpenSSL library ... which was not available for MacOS9 at the time we looked at it.

All these problems put aside, the screenshot shown in figure 5 also demonstrates how well an existing Unix/Windows Tcl application works after just a few hours of tweaking for MacOS specialities. The bigest problems were not in the application itself, but in the BWidget library used, which was never really tested for Mac.

4 Using the Tcl compiler PowerTcl

PowerTcl is written by Andreas Otto and constitutes the base of our deployment strategy. Although the ideal case of one application equals one binary is not met, compiling Tcl programs with PowerTcl supports Tcl usage in companies for several reasons:

- applications are packaged together, including a complete Tcl/Tk environment, removing the need to deploy own Tcl environments (that also could conflict with existing environments at the customers place)
- the sourcecode is not available anymore in the deployed application. It is translated into a set of compiled C instructions whereas in contrrast to that other wrappers include the Tcl code either in clear or with a easy to break encryption.
- applications are objectively and subjectively running faster once they are compiled
- cross-platform creation of binaries, e.g. Windows binaries can be compiled entirely under Linux

Currently PowerTcl works extremely well for Windows and Linux, creating code for MacOS 9 is now also possible albeit this requires a few more steps, mainly because no Unix compile environment exists that produces complete Mac binaries.

5 The trouble with simple algorithms in Tcl

One nice thing with Tcl is the ability to make prototypes completely in Tcl in very short time. On the other hand, even seemingly simple algorithms can turn into a nightmare when ported without thinking as Tcl hides the costs of most of its operations very well and most programmers simply don't know them.
While often speed is not an issue, it may become one when procs get called tens of thousands of times. There definitively *is* a difference when one has to wait 10 seconds in an interactive application instead of 5 minutes. This section shows an example how algorithms have sometimes to be rethought when used in Tcl.
Each small example will have Tcl-benchmarks for different sequence lengths and different versions of Tcl attached as example, they were made on a 550 MHz Linux machine that was not used otherwise. These numbers are not to be seen in absolute terms but provide a good possibility to compare runtimes of different programming approaches in Tcl.
It is also an excellent example on how hard it sometimes is to do the things 'right' in Tcl – knowledge of internal working mechanisms is needed. Additionally, this knowledge can become obsolete when a new version of Tcl comes out.

5.1 The GC problem

DNA is organisms consists of four bases – Adenin, Cytosin, Guanin, Thymin – which are abbreviated A, C, G and T. Those four bases concatenated form a sequence like, e.g.,

`CATGCAgtcatgTTtggtacTTGTTGttactactTGCATGCTgtactgGA`

Different measures can now be calculated on those sequence, one of these measures being the GC content, i.e. the percentage of G and C bases in the sequence. The GC content for the sequence above would be 0.42 (or 42%).
Unfortunately, there are not only ACGT in sequences, but sometimes also other characters like U (Uracil, which replaces Thymin in RNA) or so called wobbles: bases which consist not only of one type of ACGT but represent a mix of 2, 3 or all four bases. The coding for this is called IUPAC and is an international standard.

```
A=A         C=C        G=G        T=T       U=U
N=A,C,G,T
V=A,C,G     H=A,C,T    D=A,G,T    B=C,G,T
M=A,C       R=A,G      W=A,T      S=C,G     Y=C,T    K=G,T
```

It might also happen that other characters appear in the sequence, but although these count as base, they have no G or C content.
As a side condition, we expect that in more than 95% of the cases, the sequence will contain only ACG and T, and only in 0.1% we will have sequences that contain characters that are not in the IUPAC code.
The question is: how does one calculate the GC content of a given sequence in the fastest possible but safe way?

5.2 The basic C approach

This subsection deals on how a simple C routine can be ported and shows where the problems are and how these can be circumvented.

5.2.1 1:1 conversion of code

This function is almost directly ported from C except that the C code would work with a static array of 256 bytes and would not need error checking comparable to `info exists`, thus running reasonably fast as both the algorithm and the data structure would easily fit into any processor cache.

```
namespace eval GCCont_r1 {
  array set gc "
    A 0.0 T 0.0 U 0.0 W 0.0 G 1.0 C 1.0 N 0.5
    V [expr {2.0/3}] M 0.5 D [expr {1.0/3}] R 0.5
    B [expr {1.0/3}] K 0.5 H [expr {1.0/3}] S 1.0 Y 0.5
    a 0.0 t 0.0 u 0.0 w 0.0 g 1.0 c 1.0 n 0.5
    v [expr {2.0/3}] m 0.5 d [expr {1.0/3}] r 0.5
    b [expr {1.0/3}] k 0.5 h [expr {1.0/3}] s 1.0 y 0.5"
}

proc GCCont_r1::computeGCContent { sequence } {
  variable gc

  set sl [string length $sequence]
  if { $sl ==0 } {return 0}

  set sum 0.0
  for {set i 0} {$i < $sl} {incr i} {
    set c [string index $sequence $i]
    if {[info exists gc($c)]} {
      set sum [expr {$sum + $gc($c)}]
    }
  }

  return [expr {$sum / $sl}]
}
```

```
000 VERSIONS:                              1:8.4a5 2:8.4a4 3:8.3.4
067 GCCont_r1::computeGCContent 50            1061    1056    1082
068 GCCont_r1::computeGCContent 100           2104    2094    2156
069 GCCont_r1::computeGCContent 200           4194    4478    4220
070 GCCont_r1::computeGCContent 400           8223    8189    8391
071 GCCont_r1::computeGCContent 800          16618   16671   16789
072 GCCont_r1::computeGCContent 1600         33034   33107   33493
```

Unfortunately, the runtime is far from being optimal in Tcl, so let's try to improve the performance.

5.2.2 Improving the basic approach: string index performance is ... questionable

Something C programmers will find curious is that the repeated `string index` in the function above is comparably slow. It is actually faster to convert the string to a list of characters and then iterate over this list with a foreach loop:

```
namespace eval GCCont_r2 {
  array set gc "
    A 0.0 T 0.0 U 0.0 W 0.0 G 1.0 C 1.0 N 0.5
    V [expr {2.0/3}] M 0.5 D [expr {1.0/3}] R 0.5
    B [expr {1.0/3}] K 0.5 H [expr {1.0/3}] S 1.0 Y 0.5
    a 0.0 t 0.0 u 0.0 w 0.0 g 1.0 c 1.0 n 0.5
    v [expr {2.0/3}] m 0.5 d [expr {1.0/3}] r 0.5
    b [expr {1.0/3}] k 0.5 h [expr {1.0/3}] s 1.0 y 0.5"
}

proc GCCont_r2::computeGCContent { sequence } {
  variable gc

  set seqlist [split $sequence "" ]
  set sl [llength $seqlist]
  if { $sl ==0 } {return 0}

  set sum 0.0
  foreach c $seqlist {
    if {[info exists gc($c)]} {
      set sum [expr {$sum + $gc($c)}]
    }
  }

  return [expr {$sum / $sl}]
}
```

```
000 VERSIONS:                                  1:8.4a5 2:8.4a4 3:8.3.4
073 GCCont_r2::computeGCContent 50                 748     846     678
074 GCCont_r2::computeGCContent 100               1455    1623    1285
075 GCCont_r2::computeGCContent 200               2837    3180    2491
076 GCCont_r2::computeGCContent 400               5576    6280    4954
077 GCCont_r2::computeGCContent 800              10994   12556    9824
078 GCCont_r2::computeGCContent 1600             22158   25194   19836
```

This change alone cuts the runtime down by about 40%, although it seem incredible. Generating the list, storing it and recalling it from memory is faster than "direct" string indexing. How is this possible. Well, one reason for that is that Tcl stores its string internally in UTF-8 format, i.e., some characters may be represented by 2 bytes instead of one. Therefore, Tcl must always iterate through the whole string when searching a specific index, because direct string access by index could lead to a wrong result.

5.2.3 Reducing the number of array elements

Another astounding fact for people accustomed to C is that string operations are actually faster than using arrays, the less elements a Tcl array has the faster it will work. This is because Tcl doesn't know any 'real' arrays and uses hashes for this, involving hash and binary search operations. So, we're cutting the array size by half by removing the lower case bases from it and adding a `string toupper` to the proc.

```
namespace eval GCCont_r3 {
```

```
  array set gc "
    A 0.0 T 0.0 U 0.0 W 0.0 G 1.0 C 1.0 N 0.5
    V [expr {2.0/3}] M 0.5 D [expr {1.0/3}] R 0.5
    B [expr {1.0/3}] K 0.5 H [expr {1.0/3}] S 1.0 Y 0.5''
}

proc GCCont_r3::computeGCContent { sequence } {
  variable gc

  set seqlist [split [string toupper $sequence] "" ]
  set sl [llength $seqlist]
  if { $sl ==0 } {return 0}

  set sum 0.0
  foreach c $seqlist {
    if {[info exists gc($c)]} {
      set sum [expr {$sum + $gc($c)}]
    }
  }

  return [expr {$sum / $sl}]
}

000 VERSIONS:                              1:8.4a5 2:8.4a4 3:8.3.4
079 GCCont_r3::computeGCContent 50             760     850     688
080 GCCont_r3::computeGCContent 100           1641    1650    1299
081 GCCont_r3::computeGCContent 200           2864    3250    2528
082 GCCont_r3::computeGCContent 400           5545    6335    5064
083 GCCont_r3::computeGCContent 800          11102   12647   10232
084 GCCont_r3::computeGCContent 1600         22201   25386   20059
```

This will make decrease the runtime by about 3% to 5%, not much but it is a beginning.

5.2.4 Moving to integers

This is a step that most computer scientist will see as "normal", especially when having some experience in C or assembler. Using integer is almost always faster than floating point arithmetics. The weights for the bases can all be multiplied by 6 to bring them into integer, the result will have to be divided by 6 before returning to get the correct GC content though.

```
namespace eval GCCont_i {
  array set gc { A 0 T 0 U 0 W 0 G 6 C 6 N 3 V 4
    M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
}

proc GCCont_i::computeGCContent1 { sequence } {
  variable gc

  set seqlist [split [string toupper $sequence] "" ]
  set sl [llength $seqlist]
  if { $sl ==0 } {return 0}

  set sum 0
  foreach c $seqlist {
    if {[info exists gc($c)]} {
      incr sum $gc($c)
```

```
    }
  }

  return [expr {($sum / 6.0) / $sl}]
  #$
}
```

```
000 VERSIONS:                                   1:8.4a5 2:8.4a4 3:8.3.4
049 GCCont_i::computeGCContent1 50                  679     756     616
050 GCCont_i::computeGCContent1 100                1328    1435    1177
051 GCCont_i::computeGCContent1 200                2465    2805    2296
052 GCCont_i::computeGCContent1 400                4929    5568    4555
053 GCCont_i::computeGCContent1 800                9575   10972    8928
054 GCCont_i::computeGCContent1 1600              19158   22014   18113
```

While the runtime is not the great breakthrough one could hope for, it still is 10% faster than the previous version. The speedup itself is not due to integer arithmetic itself though, it is due to the `expr` being replaced by a faster `incr` statement.

5.2.5 Informations comes at a price

Looking closer at our proc and the side conditions set, one can see that the `info exists` is called for every base though we expect to have valid characters, i.e. characters that are present in the array, in almost 99.9% of all sequences.
Alas the `info exist` command is extremely costly when used on arrays, because the complete search mechanism for hashes in arrays has to be activated just to check if one certain element exists.
There is one trick to be known for cases like these: using a `catch {...}` is about 10 times faster than its `info exists` counterpart *when the catch does not fire!* If the catch fires, then the overhead for this is about 10 times greater than for the equivalent catch. As we don't expect it to fire much often, we're giving it a try, although it is looking a bit ugly and certainly does not belong to the highlights of good programming:

```
namespace eval GCCont_i {
  array set gc { A 0 T 0 U 0 W 0 G 6 C 6 N 3 V 4
    M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
}

proc GCCont_i::computeGCContent2 { sequence } {
  variable gc

  set seqlist [split [string toupper $sequence] "" ]
  set sl [llength $seqlist]
  if { $sl ==0 } {return 0}

  set sum 0
  foreach c $seqlist {
    catch {incr sum $gc($c)}
  }

  return [expr {($sum / 6.0) / $sl}]
}
```

```
000 VERSIONS:                                   1:8.4a5 2:8.4a4 3:8.3.4
055 GCCont_i::computeGCContent2 50                  271     305     242
056 GCCont_i::computeGCContent2 100                 476     539     412
057 GCCont_i::computeGCContent2 200                 859    1000     734
058 GCCont_i::computeGCContent2 400                1632    1939    1385
059 GCCont_i::computeGCContent2 800                3163    3788    2704
060 GCCont_i::computeGCContent2 1600               6290    7577    5367
```

This proc will run about 3 to 4 times faster than its predecessor, which is a big time improvement.

5.2.6 Getting exception handling out of loops

There is still room for improvements in the exception handling. Like exposed in the problem text above, in practice most sequences will behave nicely, i.e. they will not contain 'illegal' characters. We can use such a condition to further optimize the exception handling by catching the whole loop and reacting only in error cases:

```
proc GCCont_i::computeGCContent3 { sequence } {
  variable gc

  set seqlist [split [string toupper $sequence] "" ]
  set sl [llength $seqlist]
  if { $sl ==0 } {return 0}

  set sum 0
  if {[catch {
    foreach c $seqlist {
      incr sum $gc($c)
    }
  }]} {
    set sum 0
    foreach c $seqlist {
      catch {incr sum $gc($c)}
    }
  }

  return [expr {($sum / 6.0) / $sl}]
  #$
}
```

```
000 VERSIONS:                              1:8.4a5 2:8.4a4 3:8.3.4
061 GCCont_i::computeGCContent3 50             257     292     228
062 GCCont_i::computeGCContent3 100            436     510     374
063 GCCont_i::computeGCContent3 200            777     953     685
064 GCCont_i::computeGCContent3 400           1470    1819    1293
065 GCCont_i::computeGCContent3 800           2844    3613    2476
066 GCCont_i::computeGCContent3 1600          5656    7119    4906
```

This will save another 8%-10% compared to the previous version, but this also sets the limit for "conventional optimization", this proc cannot really be made any faster in Tcl.

5.2.7 A gentle introduction to playing dirty: raping some benefits of the specific algorithmic problem

In the algorithmic improvements above, one step was to convert floating point numbers to integers. One observation to be made is this: all numbers are ≥ 0 and ≤ 9, that is, these are single character numbers. Another observation made was that the cost for accessing Tcl arrays seems to be relatively high. So, is there a possibility to eliminate the array and still retain the original algorithm?

After a bit of thinking, there is one elegant solution that will appear: replace all the characters by their numbered weight equivalent and iterate over the resulting string list, just adding the numbers! Substitution is done via `string map`, which is a very fast command for this kind of operation, where one wants to replace fixed substrings with other substrings (or nothing). Exceptions, i.e., characters that are not expected in the sequence, get eliminated first by a `regsub` command, another powerful command that substitutes substrings, but this one is able to work with regular expressions. Here we go:

```
namespace eval GCCont_rsf1 {
  variable gcs {A 0 T 0 U 0 W 0 G 6 C 6 N 3 V 4 M 3
```

```
    D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
  variable gc
  array set gc $gcs
  variable re "\[^[join [array names gc] ""]\]"
}

proc GCCont_rsf1::computeGCContent sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}

  variable gcs
  variable re
  regsub -all $re [string toupper $sequence] "" sequence
  set sum 0
  foreach c [split [string map $gcs $sequence] ""] {incr sum $c}
  return [expr {($sum / 6.0) / $sl}]
}

000 VERSIONS:                                  1:8.4a5 2:8.4a4 3:8.3.4
085 GCCont_rsf1::computeGCContent 50               271     273     274
086 GCCont_rsf1::computeGCContent 100              426     413     430
087 GCCont_rsf1::computeGCContent 200              676     696     732
088 GCCont_rsf1::computeGCContent 400             1187    1264    1342
089 GCCont_rsf1::computeGCContent 800             2286    2345    2588
090 GCCont_rsf1::computeGCContent 1600            4392    4586    5002
```

This proc will run about 15%-18% faster than the last (and fastest) 'conventional' routine above.

5.2.8 Playing dirty, part 2: eliminate what you don't need

There's also another nice trick to be shown in this routine: looking at the namespace variable *gcs*, one can see that 4 characters are being replaced by the value 0. This value will later on be used in the foreach loop to increment a sum ... but why bothering to increment something by 0 at all? In a daring move, we will now replace the characters A, T, U and W with an empty string!

```
namespace eval GCCont_rsf2 {
  variable gcs {A "" T "" U "" W "" G 6 C 6 N 3 V 4
    M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
  variable gc
  array set gc $gcs
  variable re "\[^[join [array names gc] ""]\]"
}

proc GCCont_rsf2::computeGCContent1 sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}

  variable gcs
  variable re
  regsub -all $re [string toupper $sequence] "" sequence
  set sum 0
  foreach c [split [string map $gcs $sequence] ""] {incr sum $c}
  return [expr {($sum / 6.0) / $sl}]
}

000 VERSIONS:                                  1:8.4a5 2:8.4a4 3:8.3.4
```

```
091 GCCont_rsf2::computeGCContent1 50             198     198     210
092 GCCont_rsf2::computeGCContent1 100            290     276     316
093 GCCont_rsf2::computeGCContent1 200            412     430     506
094 GCCont_rsf2::computeGCContent1 400            686     717     894
095 GCCont_rsf2::computeGCContent1 800           1231    1296    1652
096 GCCont_rsf2::computeGCContent1 1600          2335    2477    3180
```

5.2.9 Again: eliminating rare case cost of exception handling

The routine above always runs through the `regsub` command to handle exceptions that we expect to happen only very, very rarely. Here again we could shave of some time by dropping the `regsub` command from the normal operation, just reacting to error situation via `catch`. There is, however, one important drawback apart from the fact that it uglifies the routine a lot. But first have a look at that routine:

```
namespace eval GCCont_rsf2 {
  variable gcs {A "" T "" U "" W "" G 6 C 6 N 3 V 4
    M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
  variable gc
  array set gc $gcs
  variable re "\[^[join [array names gc] ""]\]"
}

proc GCCont_rsf2::computeGCContent2 sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}

  variable gcs
  set sum 0
  if {[catch {
    foreach c [split [string map $gcs [string toupper $sequence]] ""] {
      incr sum $c
    }
  }]} {
    variable re
    set sum 0
    regsub -all $re [string toupper $sequence] "" sequence
    foreach c [split [string map $gcs $sequence] ""] {incr sum $c}
  }
  return [expr {($sum / 6.0) / $sl}]
  #$
}

000 VERSIONS:                               1:8.4a5 2:8.4a4 3:8.3.4
097 GCCont_rsf2::computeGCContent2 50             166     170     169
098 GCCont_rsf2::computeGCContent2 100            233     245     260
099 GCCont_rsf2::computeGCContent2 200            368     380     440
100 GCCont_rsf2::computeGCContent2 400            636     677     800
101 GCCont_rsf2::computeGCContent2 800           1163    1229    1519
102 GCCont_rsf2::computeGCContent2 1600          2215    2348    2919
```

This proc will run perhaps 2% to 4% faster than the previous routine and seems valid. So, where's the snag? Simply put: that proc will return *wrong* results should there ever by numbers in our original input sequence (which are not disallowed). Those numbers won't be substituted away by a regsub command running in front of the foreach loop, and the `incr` command will happily use any number found in the string list to add, resulting on a wrong GC content.

But we can still save that approach by enlarging the substitution set for the `string map` command, replacing any number found in the sequence by an empty sequence.

```
namespace eval GCCont_rsf3 {
  variable gcs {A 0 T 0 U 0 W 0 G 6 C 6 N 3 V 4
    M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3
    0 "" 1 "" 2 "" 3 "" 4 "" 5 "" 6 "" 7 "" 8 "" 9 ""
  }
  variable gc
  array set gc $gcs
  variable re "\[^[join [array names gc] ""]\]"
}

proc GCCont_rsf3::computeGCContent sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}

  variable gcs
  set sum 0
  if {[catch {
    foreach c [split [string map $gcs [string toupper $sequence]] ""] {
      incr sum $c
    }
  }]} {
    variable re
    set sum 0
    regsub -all $re [string toupper $sequence] "" sequence
    foreach c [split [string map $gcs $sequence] ""] {incr sum $c}
  }
  return [expr {($sum / 6.0) / $sl}]
  #$
}
```

```
000 VERSIONS:                                 1:8.4a5 2:8.4a4 3:8.3.4
103 GCCont_rsf3::computeGCContent 50              169     173     170
104 GCCont_rsf3::computeGCContent 100             240     246     260
105 GCCont_rsf3::computeGCContent 200             375     392     442
106 GCCont_rsf3::computeGCContent 400             631     669     794
107 GCCont_rsf3::computeGCContent 800            1164    1238    1503
108 GCCont_rsf3::computeGCContent 1600           2182    2431    2961
```

5.3 Activating Tcl's power: transforming strings

Tcl has become extremely efficient when it comes to transforming strings according to given rule sets – a domain where Perl until now had always an advantage – the keyword for this being *regular expressions*.
So it might be a good idea inspect the problem to see whether it can be reformulated to fit string transformation possibilities of Tcl.
The basic idea of those algorithms is not to iterate over the string and add corresponding weights for characters, but to transform the problem into a *counting* problem. Let's have a look at how to do that.

5.3.1 Beginning easily: porting the C method of doing it

We'll start first by looking how one would tackle this problem in C. What we would use there would likely be an array of 256 counters (probably unsigned int32) initialized to 0 and then go through our sequence, increasing the counters in the array at the ASCII position of the character encountered. In the end, we'd multiply each counter with the weight of the corresponding base, add everything together and would have our GC content.

This first shot in Tcl already considers some of the insights we gained up to this point, using a `string toupper` instead of a larger array and programming for the expected 99.9% of the sequences where we do not expect 'invalid' characters to appear.

```
namespace eval GCCont_cpb {
  variable countzero { A 0 T 0 U 0 W 0 G 0 C 0 N 0 V 0 M 0
    D 0 R 0 B 0 K 0 H 0 S 0 Y 0}
  array set gc       { A 0 T 0 U 0 W 0 G 6 C 6 N 3 V 4 M 3
    D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
}

proc GCCont_cpb::computeGCContent sequence {
  set seqlist [split [string toupper $sequence] "" ]
  set sl [llength $seqlist]
  if { $sl ==0 } {return 0}
  variable countzero
  variable gc
  if {[catch {
    foreach c $seqlist { incr count($c) }
  }]} {
    array set count $countzero
    foreach c $seqlist { catch {incr count($c)} }
  }
  set sum 0
  foreach {n x} $countzero {
    incr sum [expr {$count($n)*$gc($n)}]
  }
  return [expr {($sum / 6.0) / $sl}]
  #$
}
```

```
000 VERSIONS:                                1:8.4a5 2:8.4a4 3:8.3.4
001 GCCont_cpb::computeGCContent 50              618     648     709
002 GCCont_cpb::computeGCContent 100             884     948    1103
003 GCCont_cpb::computeGCContent 200            1414    1525    1887
004 GCCont_cpb::computeGCContent 400            2492    2695    3492
005 GCCont_cpb::computeGCContent 800            4686    5045    6657
006 GCCont_cpb::computeGCContent 1600           9015    9705   12967
```

Unfortunately, this approach takes approximately two to three times longer than for the best "conventional" optimization of the other C approach (GCCont_i::computeGCContent3).

5.3.2 Let Tcl do the counting

When it comes to counting characters – or, btw, complete regular expressions – Tcl has some nice and highly optimized commands for that, one of them being `regexp -all` which will return the number of occurrences of a regular expression in a string. A simple character being an extremely easy regular expression, we'll give it a shot.
Note that this time, only characters that really add to the GC content are counted, which makes exception handling of characters that are not expected unnecessary. The array has also been transformed into a list, this time it doesn't gain very much because each element is only accessed once, but anyway it looks nicer.

```
namespace eval GCCont_cpbre1 {
  variable gclist { G 6 C 6 S 6 V 4 N 3 M 3 R 3 Y 3 K 3 B 2 D 2 H 2 }
}

proc GCCont_cpbre1::computeGCContent sequence {
```

```
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}
  set sequence [string toupper $sequence]
  variable gclist
  set sum 0
  foreach {b w} $gclist {
    set num [regexp -all $b $sequence]
    incr sum [expr {$num*$w}]
  }
  return [expr {($sum / 6.0) / $sl}]
}
```

```
000 VERSIONS:                               1:8.4a5 2:8.4a4 3:8.3.4
007 GCCont_cpbre1::computeGCContent 50          455     491     445
008 GCCont_cpbre1::computeGCContent 100         677     728     683
009 GCCont_cpbre1::computeGCContent 200        1150    1199    1143
010 GCCont_cpbre1::computeGCContent 400        2047    2148    2072
011 GCCont_cpbre1::computeGCContent 800        3887    4046    3953
012 GCCont_cpbre1::computeGCContent 1600       7521    7894    7774
```

And indeed, this proc runs between 10% (for the Tcl 8.4 branch) and 30% (for Tcl 8.3 branch) faster than the C ported version above.

5.3.3 Pooling regular expressions

Having a number of regular expressions that trigger exactly the same actions – in this case, add the same value to a variable – gives the opportunity to pool those regular expressions. Again, this is done to shift work from a comparably slow Tcl command sequence (foreach loop with incr) to a relatively fast and highly optimized Tcl command (the regexp). The following proc pools all characters that have the same weight and lets `regexp` count these simultaneously.

```
namespace eval GCCont_cpbre2 {
  variable gclist {GCS 6 V 4 NMRYK 3 BDH 2}
}

proc GCCont_cpbre2::computeGCContent sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}
  set sequence [string toupper $sequence]
  variable gclist
  set sum 0
  foreach {b w} $gclist {
    set num [regexp -all \[$b\] $sequence]
    incr sum [expr {$num*$w}]
  }
  return [expr {($sum / 6.0) / $sl}]
}
```

```
000 VERSIONS:                               1:8.4a5 2:8.4a4 3:8.3.4
013 GCCont_cpbre2::computeGCContent 50          368     380     369
014 GCCont_cpbre2::computeGCContent 100         579     600     585
015 GCCont_cpbre2::computeGCContent 200        1005    1045    1019
016 GCCont_cpbre2::computeGCContent 400        1854    1933    1889
017 GCCont_cpbre2::computeGCContent 800        3560    3704    3648
018 GCCont_cpbre2::computeGCContent 1600       7028    7244    7074
```

The speed increase is between 5% and 8%, not bad for such a little change.

5.3.4 regexp vs. regsub, shootout part 1

The `regsub` command also provides the ability to count how many expressions it has substituted. The proc below uses `regsub -all` command just to count the occurrences of the character, the resulting substituted string is discarded (for now). Let's see how it compares to regexp.

```
namespace eval GCCont_cpbrs_trap {
  variable gclist {GCS 6 V 4 NMRYK 3 BDH 2}
}

proc GCCont_cpbrs_trap::computeGCContent sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}
  set sequence [string toupper $sequence]
  variable gclist
  set sum 0
  foreach {b w} $gclist {
    #set num [regsub -all \[$b\] $sequence "" sequence]
    set num [regsub -all \[$b\] $sequence "" discardme]
    incr sum [expr {$num*$w}]
  }
  return [expr {($sum / 6.0) / $sl}]
}
```

```
000 VERSIONS:                                1:8.4a5 2:8.4a4 3:8.3.4
037 GCCont_cpbrs_trap::computeGCContent 50       412     424     446
038 GCCont_cpbrs_trap::computeGCContent 100      682     656     689
039 GCCont_cpbrs_trap::computeGCContent 200     1079    1106    1175
040 GCCont_cpbrs_trap::computeGCContent 400     1959    2021    2142
041 GCCont_cpbrs_trap::computeGCContent 800     3725    3837    4052
042 GCCont_cpbrs_trap::computeGCContent 1600    7253    7479    7884
```

The resulting run time of this change is not quite convincing, but understandably a bit slower than the regexp example as it has to build and save a string each time it goes through. Even when reusing the new string as shorter sequence for each iteration, the runtime is still slower than the regexp version.

5.3.5 regexp vs. regsub, shootout part 2 (differences in Tcl 8.3 and Tcl 8.4)

Taking a step back in our thoughts, we're going to try the same proc again, this time with each character having again it's own weight.

```
namespace eval GCCont_cpbrs {
   variable gclist { G 6 C 6 N 3 V 4 M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
}

proc GCCont_cpbrs::computeGCContent1 sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}
  set sequence [string toupper $sequence]
  variable gclist
  set sum 0
  foreach {b w} $gclist {
    set num [regsub -all $b $sequence "" bla]
    incr sum [expr {$num*$w}]
  }
  return [expr {($sum / 6.0) / $sl}]
```

```
}

000 VERSIONS:                                   1:8.4a5 2:8.4a4 3:8.3.4
025 GCCont_cpbrs::computeGCContent1 50              255     260     674
026 GCCont_cpbrs::computeGCContent1 100             287     296     968
027 GCCont_cpbrs::computeGCContent1 200             368     373    1520
028 GCCont_cpbrs::computeGCContent1 400             490     510    2598
029 GCCont_cpbrs::computeGCContent1 800             756     815    4776
030 GCCont_cpbrs::computeGCContent1 1600           1298    1350    9156
```

The result from running this proc will be mixed: for the Tcl 8.3 branch, the runtime will be even slower than the previous version. The astonishment will come when running an 8.4 branch: the runtime against the regexp version will be divided by at least a factor of 2 for short sequences and and will go well beyond a factor of 5 for somewhat longer sequences.
We can improve the runtime even further a little bit by reusing the regsub sequence in each iteration, the shortened strings gets made anyway:

```
namespace eval GCCont_cpbrs {
   variable gclist { G 6 C 6 N 3 V 4 M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
}

proc GCCont_cpbrs::computeGCContent2 sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}
  set sequence [string toupper $sequence]
  variable gclist
  set sum 0
  foreach {b w} $gclist {
    set num [regsub -all $b $sequence "" sequence]
    incr sum [expr {$num*$w}]
  }
  return [expr {($sum / 6.0) / $sl}]
}

000 VERSIONS:                                   1:8.4a5 2:8.4a4 3:8.3.4
031 GCCont_cpbrs::computeGCContent2 50              239     251     731
032 GCCont_cpbrs::computeGCContent2 100             270     279    1012
033 GCCont_cpbrs::computeGCContent2 200             330     345    1603
034 GCCont_cpbrs::computeGCContent2 400             445     463    2737
035 GCCont_cpbrs::computeGCContent2 800             677     724    4950
036 GCCont_cpbrs::computeGCContent2 1600           1137    1205    9450
```

This saves a few percent runtime.
As last optimization in this branch, we reuse our knowledge of the expected data, namely that the bases A, C, G and T will appear predominantly and that we should try to concentrate on these. Two things change for that in the following proc: 1) the input sequence gets cleaned via the fast `string map` from the frequent A and T bases (taking U and W with does not harm the runtime performance noticeably) and 2) we have 2 lists with weights for the GC content, the first containing only the weights for G and C, so that we hope that after this step we won't need to go through the second list as the sequence will be completely worked through if it contained only A, C, G and T.

```
namespace eval GCCont_cpbrs2 {
  variable gclist1 { G 6 C 6 }
  variable gclist2 { N 3 V 4 M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
}
proc GCCont_cpbrs2::computeGCContent sequence {
  set sl [string length $sequence]
  if { $sl ==0 } {return 0}
```

```
  set sequence [string map {A "" T "" U "" W ""} [string toupper $sequence]]
  variable gclist1
  variable gclist2
  set sum 0
  foreach {b w} $gclist1 {
    set num [regsub -all $b $sequence "" sequence]
    incr sum [expr {$num*$w}]
  }
  if {[string length $sequence]} {
    foreach {b w} $gclist2 {
      set num [regsub -all $b $sequence "" sequence]
      incr sum [expr {$num*$w}]
    }
  }
  return [expr {($sum / 6.0) / $sl}]
}
```

```
000 VERSIONS:                                1:8.4a5 2:8.4a4 3:8.3.4
019 GCCont_cpbrs2::computeGCContent 50           153     154     400
020 GCCont_cpbrs2::computeGCContent 100          186     190     652
021 GCCont_cpbrs2::computeGCContent 200          247     257    1171
022 GCCont_cpbrs2::computeGCContent 400          366     381    2202
023 GCCont_cpbrs2::computeGCContent 800          612     638    4231
024 GCCont_cpbrs2::computeGCContent 1600        1120    1166    8293
```

This optimization brings most for short input sequences (up to 40%) and not too much for longer sequences (down to a few percent).

5.4 Getting crazy: algorithms you'd never thought of in C

The following examples are examples on how thinking on sidetracks can sometimes lead to some astonishing results, beside the fact that these algorithms work, that is.

5.4.1 I'm a Turing Machine ... sort of

It is sometimes quite easy to transform parts of an algorithm in its equivalent Turing automaton. For example, all what we really want to know from the sequence is the (weighted) number of Gs and Cs compared to other bases. In the process of optimizing all those algorithms above, we already collected all the ingredients needed to simulate the output of a simple Turing machine on the sequence: we work with weights that are integer and have a method to replace characters in a string with other characters.

Imagine the base sequence to be our input alphabet. Now let the TM start at the beginning of the sequence and give it the command to replace every letter of the defined input sequence by the number of characters of the equivalent weight, don't care about unknown characters. The just count the length of the resulting string of the output sequence.

For reasons explained a bit later, we'll take the character $>$ as output character. Let's look at the translation table

```
Input     Output

A
C         >>>>>>
G         >>>>>>
T
V         >>>>
M         >>>
D         >>
...
etc.
```

Here is one example how the translation would be made (spaces added for better readability, empty replacements shown by “”):

```
ACCTMGAD =  "" >>>>>> >>>>>> "" >>> >>>>>> "" >>
```

which results into a string with a length of 23 characters, so that $23/6/len_{input} = 23/6/8 \approx 0.479$
Our real world implementation will work with a subtle difference because of the limitations of the `string map` command that is used: there is no way to to tell string map to replace unknown characters with nothing, it just leaves them untouched. The algorithm will therefor first simulate a Touring Machine that leaves unknown characters untouched just to eliminate them in the following `regsub` command.

```
namespace eval GCCont_turing {
  set list { A 0 T 0 G 6 C 6 U 0 N 3 W 0 V 4
    M 3 D 2 R 3 B 2 K 3 H 2 S 6 Y 3}
  variable map ""
  variable mapnc ""
  foreach {letter num} $list {
    set code [string repeat > $num]
    lappend mapnc $letter $code
    lappend map $letter $code [string tolower $letter] $code
  }
}

proc GCCont_turing::computeGCContent { sequence } {
  variable map
  set sl [string length $sequence]
  if {$sl == 0} {return 0}
  set pstr [string map $map $sequence]
  regsub -all {[^>]} $pstr "" pstr2
  set sum [string length $pstr2]
  return [expr {($sum / 6.0) / $sl}]
}
```

```
000 VERSIONS:                               1:8.4a5 2:8.4a4 3:8.3.4
109 GCCont_turing::computeGCContent 50          120     121     168
110 GCCont_turing::computeGCContent 100         165     140     252
111 GCCont_turing::computeGCContent 200         195     191     409
112 GCCont_turing::computeGCContent 400         286     284     735
113 GCCont_turing::computeGCContent 800         478     472    1389
114 GCCont_turing::computeGCContent 1600        849     846    2649
```

When comparing the runtime of this algorithm to the previous designed ones in Tcl, one will probably get the shock of the life: it’s actually faster that any other presented above (especially with the Tcl 8.4) branch.

5.4.2 Nice ideas that backfire, part 1

A nice substitution could be to replace the whole sequence of letters so that a mathematical expression arises that can be evaluated by `expr`. The following code will do just that, it is small and nice and deals with unknown characters via `regsub`.

```
namespace eval GCCont_expr {
  variable gclist {A "" T "" U "" W "" G 6+ C 6+ N 3+ V 4+
    M 3+ D 2+ R 3+ B 2+ K 3+ H 2+ S 6+ Y 3+}
}
proc GCCont_expr::computeGCContent seq {
  set l [string length $seq]
  if {!$l} {return 0}
  set ns [string map $GCCont_expr::gclist [string toupper $seq]0]
```

```
  regsub -all {[^0-9+]} $ns "" ns2
  return [expr ($ns2)/($l*6.0)]
  #$
}
```

```
000 VERSIONS:                              1:8.4a5 2:8.4a4 3:8.3.4
043 GCCont_expr::computeGCContent 50           279     286     288
044 GCCont_expr::computeGCContent 100          448     457     462
045 GCCont_expr::computeGCContent 200          859     916     918
046 GCCont_expr::computeGCContent 400         2119    2301    2231
047 GCCont_expr::computeGCContent 800         8435    8879    8409
048 GCCont_expr::computeGCContent 1600       38953   42026   40019
```

Unfortunately, the runtime is well below acceptable terms, especially for longer input sequences. This is because internally, `expr` has to build an expression tree for determining how to interpret the expression, and this gets quickly ugly for long expressions, the runtime grows exponentially.

5.5 Wrapup

The speed increase factor between the initial C ported routine and the Touring Machine based Tcl construct lies between 7 to 9 for small sequences (50 bases) and 12 to 40 for longer sequences (1600 bases). This shows how important fine tuning in certain areas can be.