

SHAPE JOURNAL

PROGRAMMING TODAY

FROM WHAT GROUND SHOULD WE CRITICISE IT - NOTES ON CODE RED / RED FOR DANGER
PAINTING BY NUMBERS / C++ AND THE PHILOSOPHY OF MATHEMATICS / TOO MANY NOTES



**Shape Journal
Issue 31**

Programming Today:

- 1. Introduction**
- 2. Code Red for the Users?**
- 3. Red for Danger**
- 4. C++ and the Philosophy of Mathematics**
- 5. Painting by Numbers**
- 6. Too Many Notes**
- 7. Postscript: Diversionary Motives?**

Data.tron [3 SXGA+ version] (2007-09)
by Ryoji Ikeda

Issue 31

©2013 Jim Schofield
Words Jim Schofield
Design Mick Schofield

www.e-journal.org.uk/shape

Shape Journal
Bild Art
11a Woodlands Road, Lepton
West Yorkshire. HD8 0HX UK

Introduction

Notes on Code Red



Welcome to issue 31 of the **SHAPE Journal**.

This is another unusual set of papers! And to address the problems involved, the author has had to include several fairly long-in-the-tooth articles that were, in their time, and still are, among the clearest that are available concerning key issues involved in Programming Languages.

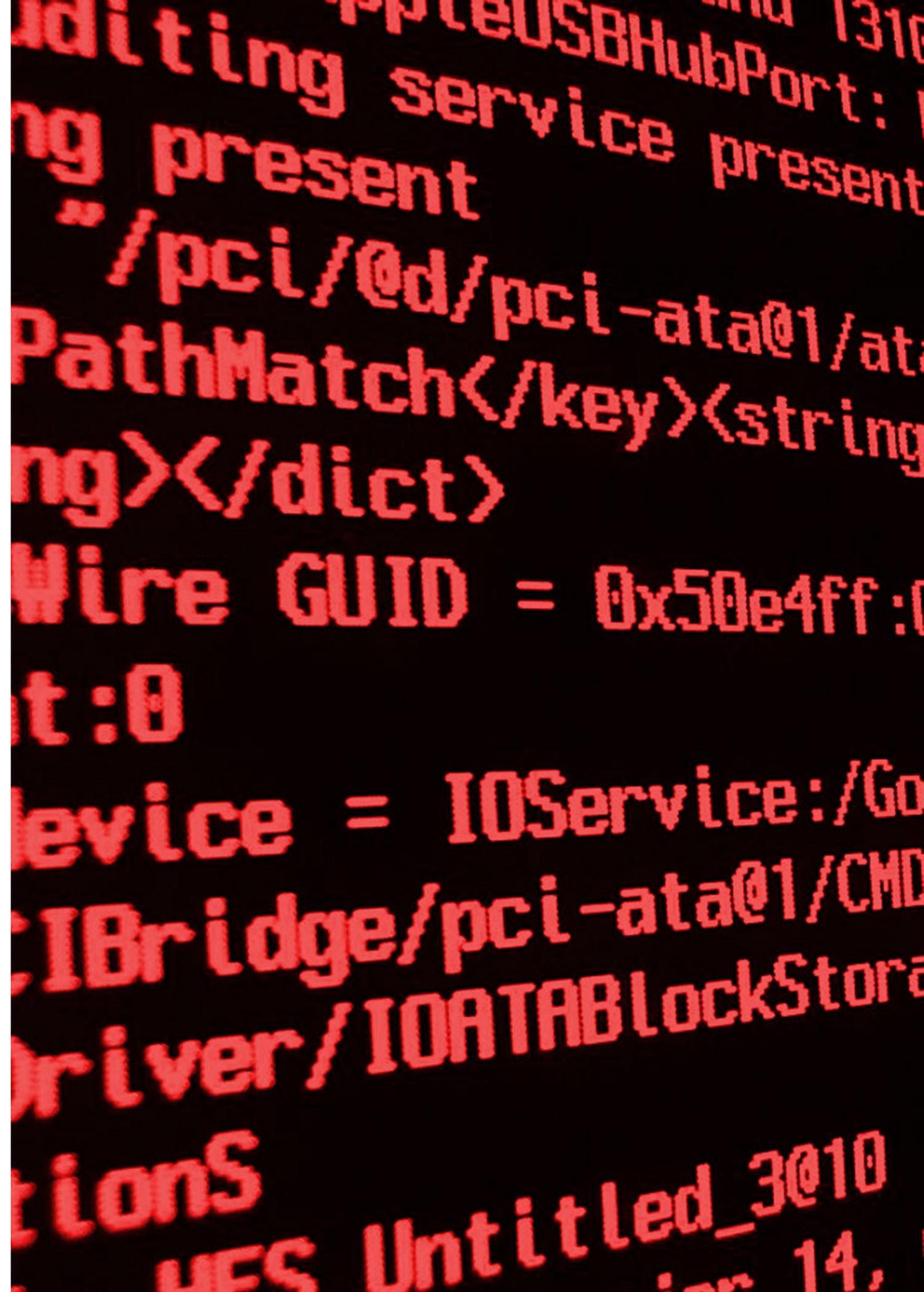
There is one from 1998, and another from 2003, and I have to admit that several topics raised in these papers show their ages, via many of the pieces of ancillary kit and software packages that are no longer in use (or even remembered). But, these historically-defined details do not, in any way, undermine the general points made, for the causes for these diversions are still with us to this day. So, the dated references have not been removed or replaced.

In addition, in discussing the pros and cons of programming that were raised in Michael Brooks article *Code Red* in New Scientist (2920), it became evident that Computer Languages, as such, were unavoidably imbued with certain incorrect, and diverting assumptions carried over mainly from Mathematics (and to a lesser extent from Science itself), which certainly guarantee that certain vital natural processes and indeed, developments, are unobtainable, within the confines of these entirely formal means.

So a short paper has also been included, that at least begins to address this problem in some detail too.

The historical papers on C++ and Flash (Actionscript), may be referring to older versions of these systems, but the points made are still to this day, entirely valid, and I also couldn't just ignore Michael Brooks' evident lack of any real understanding in the area of programming and Programming Languages, without delivering a strong criticism of his position. Publishing these papers has allowed me to do just that.

Jim Schofield July 2013





Code Red for the Users? From What Ground Should We Criticise I.T.?

This is a first detailed response to the Michael Brooks article “Code Red” in New Scientist (2020), concerning crises in computer software. There is also a second paper, which takes a more philosophically critical line under the title “Red for Danger: Beware the I.T. specialist”, which takes a critical stance from the inside by an ex-Director of Information Technology at London University.

The first thing to make absolutely clear, about this article, is that Brooks knows nothing about the creation of any of the many kinds of software produced by qualified professionals in this particular area. He is, at best, an amateur user of it as a replacement for his typewriter, and thus he is in no position to be able to fathom the crises that have in the past, and still to this day, beset this increasingly important area of human activities. I can only compare his uninformed judgements with those of politicians drastically re-organising Education, with a similar lack of the requisite knowledge to do anything but make it worse.

Hence his contribution has to rely upon fragments or one-liners from those who do have the necessary expertise, or from other users like himself. And as a professional in this area myself, I can say that all the experts quoted from were clearly “computer nerds”. Their discipline is really all they know, and their expertise is in “extending” and “improving” that body of knowledge and techniques. And, as with mathematicians, who do the very same things in their own realm, they merely develop the techniques involved in their own terms alone.

They do NOT import ever-new problems, requiring new solutions to widen and deepen their discipline. They exploit the ever-new, technological advances, of quite a different group, who are basically electronic engineers, to “solve” the problems of others, incapable of doing it for themselves – but only if the techniques are already in their armoury.

I always remember going for interview for a post in a Scottish University, and was asked by the interviewer, “What is your problem?”. I assumed he wondered what was causing me difficulties, but, of course, he meant what was I researching within Computing. When I understood what he was asking, I explained that I was primarily interested in inter-disciplinary developments, where an intrinsic knowledge of the problems to be addressed in other disciplines, could only be tackled by computer specialists willing to subordinate their own “problem areas” to serving the detailed needs of the other discipline, for, in my experience, the most profound developments in I.T. were invariably achieved in such situations.

But, my interviewer was singularly unimpressed. Though as the following 25 years was to prove, he was significantly mistaken!

Clearly, the biggest problem in I.T. is the inwards-turning, and indeed parochial attitude of those within that discipline. They treat all problem-solving techniques as “general” – as independent of the area of application, and THIS is, without doubt, the most crucial source of their difficulties.

This deflection of appropriate lines of development, and a preoccupation of “doing something new” in computing, has led to Brooks’ main problem – “Which language do I choose to learn?” He explains that he cannot learn to program until he knows which of thousands of computer languages he should conquer. But, that is a perfect example of “the cart before the horse”. Choosing the language cannot come first!

What is programming for? Surely, it is to employ the power and speed of a computer to tackle a difficult, time-consuming or onerous problem. Once you have your important problem, your choice will be made for you. For, if you know what it is that you wish to use, in an area important to you for other reasons, you then search for a program in that precise area, either doing exactly what you want, or something very close. Having found appropriate programs, you THEN know, immediately, which language you will have to learn – for a majority of your finds will be in it. All the exemplars you unearth in your search are in an area you know about, and are confronted with finding a large number of programs to study.

Experience has proved that such a starting point, makes learning the language involved very much easier, AND crucially, having got one of the revealed set to work, the adjustments required to make it deliver exactly what you require will be much easier. In no time at all, you will have written a successful program, based upon someone else’s initial form, but tailored by you to do what you need. Thereafter, you tackle new problems and with appropriate searches find helpful examples to deliver what you need.

Surprisingly quickly, you are conceiving of programs of your own, and learning new features of the language as you go. The excuse of too many computer languages is a get out!

Also, the venom Brooks finds on the Internet, about the best and worst computer languages, is no good reason for doing nothing. All such languages will do significant things. Until you actually try to implement something, you will never be in a position to judge. For, most of that venom is parochial – “I know this language, and all the others are rubbish” – or the most unhelpful of all – “They are all rubbish!”

Using an ancient language - **Algol**, I quickly found that I could do literally anything. And the various versions of BASIC are quite adequate for most tasks.

In 1989 I won a British Interactive Video Award – a UK National award, for a multimedia package written for a BBC B Computer using BASIC to control what was then the latest technology Video material on a Philips Laser Disc.

So, just find a program that is concerned with something you know about, and attempt to understand it. It is much easier than you think!

Finally, Brooks feels that all languages are poorly designed, so it is impossible to get into them. Again, Not so! And, for some reason he is talking about the languages mostly used in modern Systems Programming – the most difficult area: and NOT the place to start! He clearly doesn't know about languages that are similar to spoken languages – the so-called Procedural Languages. And the low-level primitives that are utilised by all higher level languages.

In 1989 I used the widely-employed, Interpretive Language, BASIC, but wedded to a library of “mouse primitives” off the Internet (very easy to use, and supplied with full instructions) and with these implemented a program that BASIC (at that time) couldn't do. Indeed, the best compilers and interpreters now allow insertions of machine code or assembly language primitives, within an easy, high-level, main program.

I used to teach Programming at a Further Education College via a year-long course that cost just a few pounds, and literally everyone could program something that worked well within just a few weeks. You build up your knowledge by concentric subsets as further facilities become necessary.

Stay well away from “codey” languages, for they are designed expressly for computer nerds, and are frequently impenetrable for non-specialist programmers!

Ordinary mortals, initially at least, just want to solve a problem, so procedural languages are best.

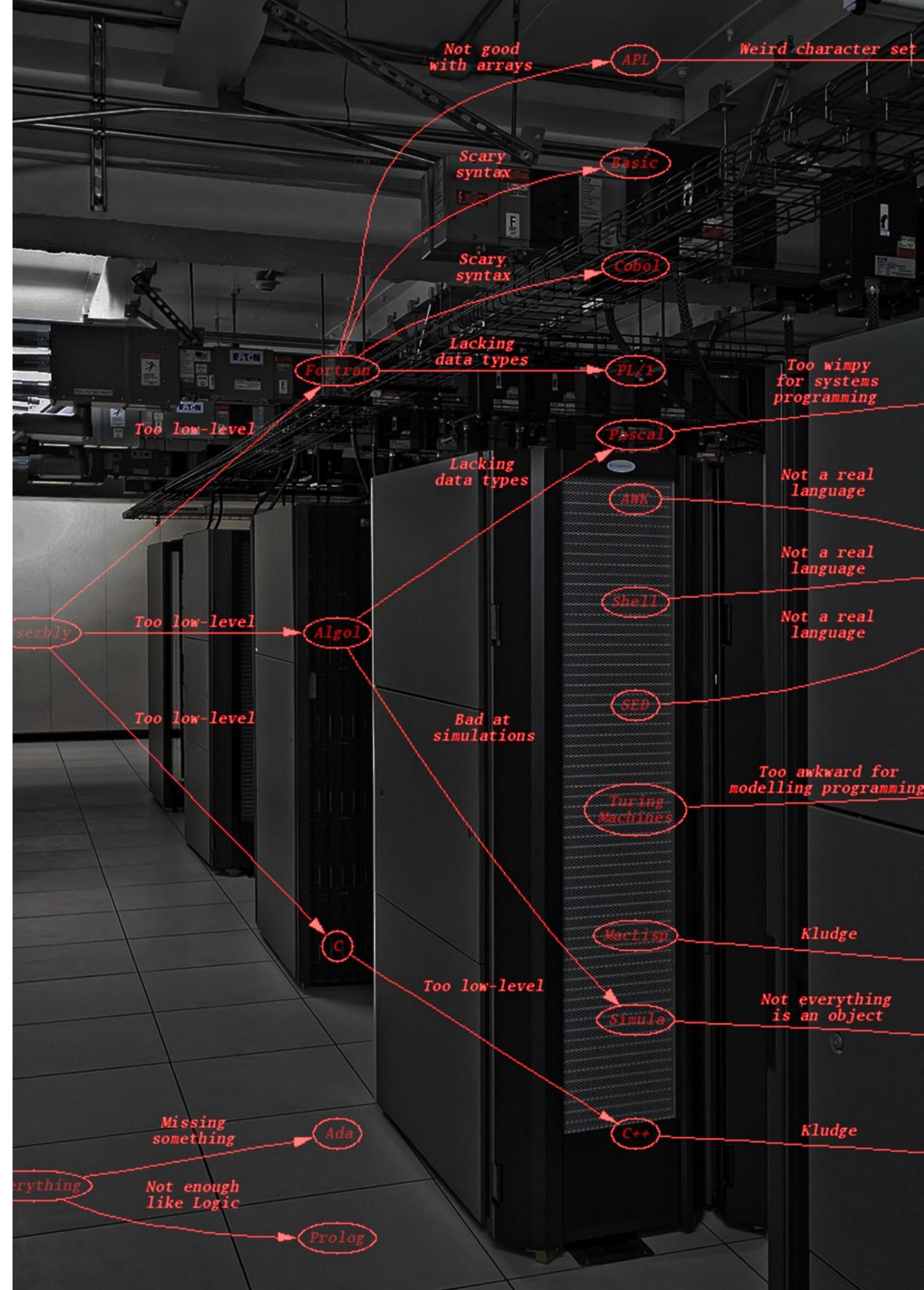
Brooks next addressed the tragedy of modern programming, which our uninformed commentator puts down to the “poor nature” of programming and programming languages. Once more, this is NOT the case! He omits the lazy cut-and-paste approach, wherein programs are essentially merely collages of other peoples coding. Sometimes, of course, you cannot avoid it, as what you offer up to a compiler or interpreter is usually replaced by blocks of ready-made code, of which you don't have any idea what the programmer (or programmers), who wrote it, had in mind. In the case of so-called interpreted languages, and those with plug-in libraries, this is the norm. The majority of any program that you may write will make your personal contributions seem like a winkle-on-a-whale of prewritten blocks, sometimes of enormous size. Debugging what you write may not solve an evident problem. It is likely to be someone else's code, not only inaccessible to you, but also uninterpretable by you, if revealed.

Well, what caused this mess? You did! By wanting to program without doing it all yourself. The writers of systems code attempted to do it all for you - it would be their code that actually did the work. Your enclosing “shell” program merely accessed these enormous blocks of code. It was this attempt at making the writing of programs easy that meant you couldn't adjust anything below your superficial “calling shell”, sitting on top!

The argument for the proliferation of languages must therefore be put down to the variety of applications, and the tailoring of languages to particular sets of problems, in a given area. And, this was a positive reaction to the repeatedly failed attempts to produce a single, general, do-everything language, which was good enough to answer everyone's needs. For, to attempt to deliver that would require enormous amounts of already-written blocks of code.

The example of the language PL1 is usually put forward as just such a do-everything language, but it, and others of the same ilk, always failed by being too big: to cover absolutely everything, the number of commands and forms, not to mention ready-made code had to be so considerably increased, as to make the language, and its various ancillary subsections and libraries, both much too big, and much too slow. It would include multitudinous tests for options that would never arise in any particular limited purpose for individual programs written in it.

So, the culture switched over to the direct opposite, and new languages, dedicated to a particular area, would, henceforth, be tailor-made for it, and these proved to be much smaller and faster, and delivered a limited set of options extremely quickly.



Thus, many of the proliferation of languages were produced for this very sound reason, and you certainly don't have to choose from the enormous overall set at all. Querying a search engine to find something like what you need, will deliver solutions written in a much smaller set of languages, designed to tackle your very type of problem.

But, it must also be admitted, that many of the enormous proliferation of languages are produced to earn doctorates involving entirely original research, and, of course, nothing is ever as original as an entirely new language. Yet even this will be clarified by how many people thereafter use the new language. In the suggested search for programs that may do your particular job, you will see large numbers in one or two languages, and just one or two in each of many other languages. You, of course, go for the popular languages in your area. The task of which language to use shrinks by the minute!

An example, of how complicated this is supposed to make programming, is given in the Michael Brooks article, by a certain Alex Payne, when he informs us that "Facebook alone uses C++, Java, PHP, Perl, Python and Erlang among others". But, this is a red herring: decent programming systems universally allow the inclusion of code in other languages, which have already solved common problems, and which can merge them in seamlessly, as long as the communicating parameters are clearly known and appropriately presented. Indeed, it is because they make programming much easier, that such a mix has been made possible. But, this does mean that more and more of your program was actually written by someone else: someone who you have absolutely NO access to.

The supposed hidden errors in those "foreign blocks" of code, are supposed to represent a major problem, but they are included so that the programmer didn't have to do it for himself. And, it must be said, that most of these are so universally used, that they are after quite a short period, are made particularly free from errors.

Of course, if the programmer (looking for an easy path) included the wrong optional code, it may well not do what he thinks it will, and the supposed error in the code, could well turn out to be the error of using that option lazily, and without due care, checks and even test programs to clarify exactly what it does.

But there are also other things to be considered. All High Level Languages take the underlying primitive functions as read: they are hidden within the code, and are not usually accessible from the high level itself. But, in a world in which new things need to be controlled, with input devices such as mice and touch-screens, and even to very unusual controlling of external systems delivering Streaming Video footage and many more (particularly in my own specialism of Computers in Control), which necessarily involve the patching-in of many new primitives,

these require a language that can do this, and hence provide a means for the high level forms to access and use such required new blocks of low level code.

Some of these can be done for you, and made available as libraries. But, the real High Level Languages have to have the methods to integrate and access these. The problem is always to attempt to do everything within a single language, and this imperative inevitably led to languages that were intended to do so. But, they became much less fathomable and much more codey in consequence. Finally, the problem could never be effectively solved by a single, do-everything language. Such ambitious constructions invariably got too big, and gobbled up vast amounts of computer memory. And, in addition they can only be the products of many hands and extensive teams, and led to "the left hand not knowing what the right hand doeth". Too much is unknown about the overall package, for the writer(s) of a particular update, or correcting patch, for the full consequences to be understood, and therefore guarded against.

It is like the Theorems of Mathematics, which people try to remember, and invariable have misconceptions along with the bits they get right. As a professional mathematician myself, I never even tried to remember them all, but I knew how to derive them. Even in exams, I always used to derive them from scratch, and in so doing, not only got the required formulae right, but also so familiarised myself with the area, that the use-part of the set problem was straightforward to deliver.

But, you can't do that with modern-day-programming: there are simply too many, along with insufficient skills for everyone to do that. Often, it is better to do as much as possible yourself to avoid these difficulties. And this does work! Believe it or not, I could develop a package from scratch in a fraction of the time that was required to achieve the same outcome by a team, and knowing every part of what I had written, debugging and safely correcting errors was always achievable.

Now, Brooks' article attempted to illustrate the possible solutions to all the above-described difficulties of computer languages. The initially suggested answer was to lay down a Master, Do-Anything Language, but the usual way of arriving at a consensus – The Academic Conference, failed miserably to come up with a universally agreed design. For one thing, such an undertaking would scupper ten thousand half finished doctorate theses, and not a few careers. Such a Conference always became a battleground! Yet, when just such a Conference was convened to decide on the form of such a language, it descended, as usual, into the same parochial chaos, and was guaranteed to be yet another failure, until one group revealed that they had already produced what was required, which was called Algol. Immediately, everyone scrambled to try out the new form.

Why do you think that so many important gains are made by the military?

It is because they know exactly what they need, and lay it out in the clearest possible way. The producers don't do what they think the military need: they do what the military say. It isn't that the military are better; it's just that they are in sole charge, and refuse anything that doesn't deliver to their specific definition. And, in my specialist area – Computers in Control, working with a wide range of disciplines, it was exactly what the users required, which determined what was sought by them, and delivered by me. So it's no good asking those who are trying to establish their status within the Information Technology area: you must ask the users, and after you have informed them that it could-not-be-done, you then had to set about delivering it. NOT, as is usual, with some already existing technology, but instead working out exactly what facilities they needed, and finding wholly new ways to deliver them.

But, such a route was usually abandoned, and one possible solution to the problem of finding and correcting errors (considered to be a much more important task) was one similar to the gobbets of programming that occur in the cells within Spreadsheets. If a language was produced that had a similar overall form corrections and adjustments would be considerably easier to cope with. For the cells in such packages, can not only contain numerical values, or character strings, but also small pieces of code referring to the contents of other cells, so that when new data was entered into the appropriate cells, the code in another cell, which refers to these, is automatically and immediately updated, so you can see immediately the effect of your new entries. Similarly, changes in the code in a cell, will immediately deliver the effect upon data by that change.

Thus, a new language, or a new compiler for an existing language, but supplied with such a cell-structure, so that all changes would be available as soon as they had been included. It would be like a built in Trace, and checking the accuracy of your new code would be considerably easier. But, not all programming can be force-fitted into that kind of template. None of mine, always involving in-and-out communications with external kit would ever fit that paradigm.

Jonathon Edwards (MIT) has started upon such a language (for web applications), which he calls Subtext. It remains to be seen how successful he will be.

And, several other researchers have attempted similar means of showing the consequences of changes as they are made. Chris Granger's Light Table effectively gives easy access to all consequent effects (as if they were all on your desk in front of you, and updated as you make changes.) One proposed solution was for computers (via an interpreting piece of software) to write the programs themselves in response to a series of questions posed by

the person requiring a program. The person who suggested this was Christa Lopes, who asserted that the answers to all such questions are there already.

I'm afraid not!

It could never work for a series of reasons. First all the suggested FAQ responses can only answer with the most commonly appropriate answers, and these will not always be correct.

Secondly, on what basis would these be linked into a program? It could only be based upon Formal Logic and the assumption of the Principle of Plurality.

Finally, the overall purpose could never be communicated, and hence no defined context could be inferred.

You would get a program, but the chances of it doing anything NEW would be absolute zero. It could only be entirely retrospective and with a Lowest Common Denominator remit.





Red for Danger Beware the I.T. Specialist!

Mistakes in the article in New Scientist (2920) entitled Code Red by Michael Brooks, were clearly signalled, amazingly, in excerpts from its first and last paragraphs which admitted:- “It’s time I taught myself how to program”, and terminally, “There’s hope for me yet!”

I’m afraid not, Michael, for to program a computer is not merely an acquired skill, for to do anything at all worthwhile, you have to be a very different animal, than one who only knows how to code situations that he can do by hand into a given Computer Language. For such are termed “hacks” and given the very lowest ranking and payment in computing circles. The next layer up are the Systems Analysts, who designed the packages that made possible the delivery of effective, practical programs, While the top experts were considered to be those who designed what are termed Systems Software – Operating Systems or Computer Language Compilers (translators into machine code).

BUT, beyond all of these Pure I.T. skills, there is another higher calling: they are the consultants who know (or know how to find out) the imperatives and indeed the necessities of individual disciplines to be served NOT in the usual generalist way, but by means of wholly new functions that computers, for the very first time have made it possible to address.

This computer expert (I finally achieved the post of Director of Information Technology in a College of London University, went through all these layers, and ended up in the type of role I have just described as the highest and most demanding) DID NOT merely apply generalist methodologies to various disciplines, but instead immersed himself into each discipline, so deeply, that he began to devise discipline-dependant functions and the facilities to carry them out.

For such turn out to be very rare animals indeed, and are very unlikely to be of the same ilk as most Computer Specialists. For, they must always be the respectful servers of the discipline that requires their aid, and no common or garden computer techie will ever fit the bill. For, they will be too general in their approach, and treat all disciplines alike as requiring the same powerful general packages. On the contrary, these experts will find wholly new features and facilities in the served discipline, which would never be discovered by know-it-all generalists.

But, as can be seen from the above paragraphs, the insider-ranking system within computer departments and in the usual, generalist-serving companies, have time and again

proved wholly inadequate to special disciplines, which are capable of generating entirely new functions, and need people capable of creating them. The parachuted-in, generalist computer-buffs or even self-taught amateurs have proved to be hopelessly constrained in what they can deliver. The real stuff can only be delivered by those with a wide experience in many very different disciplines, who know that they must first make themselves subservient to the discipline experts, will spend a great deal of time understanding the objectives and methods, and only then gradually match what they profoundly understand in their own area to that of the discipline experts, and then together develop wholly new and radically transforming I.T systems based soundly upon the core of the served discipline.

Now, my wide experience of Information Technology Departments in Hong Kong, Glasgow and London, has made it clear that such consultant experts are NOT produced.

Perhaps, I should reveal myself, having moved about quite a bit in a career (in computing) of well over 40 years, I spent several decades aiding and abetting Higher Education discipline experts, who sorely needed tailor-made software to facilitate, and indeed develop, the research they were doing in their specialist areas.

Such researchers already knew what they needed to do, but it generally never involved computers, nor the usual administrative sets of Office-type packages. They were not what was required.

Now, retrofitting of Databases, Spreadsheets, or even Communications Packages would simply NOT suffice. They invariably wanted something that was particular to their discipline and current methodology, and no visiting “expert” could ever furnish what was really required.

The usual prototype of “helpful intervention” was demonstrated very clearly in Dance, when “helped” by a Computer Science Department from a prestige University in Spain. Having seen the efforts of this expert (Jim Schofield) the Spanish Computer Department were convinced they could do much better, and a substantial team led by senior members of the Department was assembled to “show how it ought to be done”. They knew all the main functional areas and how the write efficient code to deliver them, and as far as they could see, it was merely a task of fitting their templates to this department’s needs.

But, it didn’t work: none of the programs that were delivered addressed the real problems of the Dance Department.

What they did produce involved all sorts of up-to-the-minute-methods, and allowed all sorts of manipulation of video materials. But, they had no idea of what the Dance people wanted to do with the footage, and addressed none of the difficulties of the staff members, student dancers and certainly gave zero aids to choreographers. The key tasks of presenting, interpreting or even designing appropriate movements were ignored. The most basic learning of correct and expressive dance movements were impossible with the facilities that the I.T. experts delivered. "I can't see what is happening with her left elbow in your footage", dancers would justifiably complain. "How am I supposed to balance? For the transition into that movement is simply unclear" And also you would frequently hear, "The orchestration of her limbs, and the consequent transition into the next phrase is ambiguous" "I can't tell what I am supposed to do: should I just try various alternatives, and choose what I think is best?" No!

In interpreting an exemplar piece, and indeed modifying it due to inadequate information, you are no longer learning from the experience of the choreographer, but inserting alternatives from the student's much more limited experiences. You can't do that!

Clearly, these computer experts had no idea what the real problems within the discipline were. Had they never watched the usual methods of animators communicating what they actually knew from previous performances of the same piece? The I.T. product, which they produced, in spite of its brilliance as a piece of programming, was useless as an aid to teaching Dance, its intended function.

And, if the reader perhaps thinks that Dance is a very special case then I must disagree. In a period of 10 years, aiding discipline experts from many different areas, I can say that such problems were indeed the norm. It was only a very basic set of "admin-routines" that could be addressed in the way these experts were doing.

The use of computers (especially in control), to aid in almost all disciplines, could only be achieved by learning sufficient of that discipline, and constantly allowing the experts to complain and disagree, that the correct and effective marriage could ever be brought to consummation.

The Spanish software justifiably vanished without a trace.

Now, that crucial aspect was not all that was necessary to characterise the ideas and products of Computer Studies Departments. Rather than becoming flexible and wide students of all disciplines, so as to be in a position to make a significant contribution, the members of such a Department felt that they had to make their contributions to computer software as such. And, if you limit your "telling contributions" in that way, you are very likely to invent and then implement a "better" Computer Language.

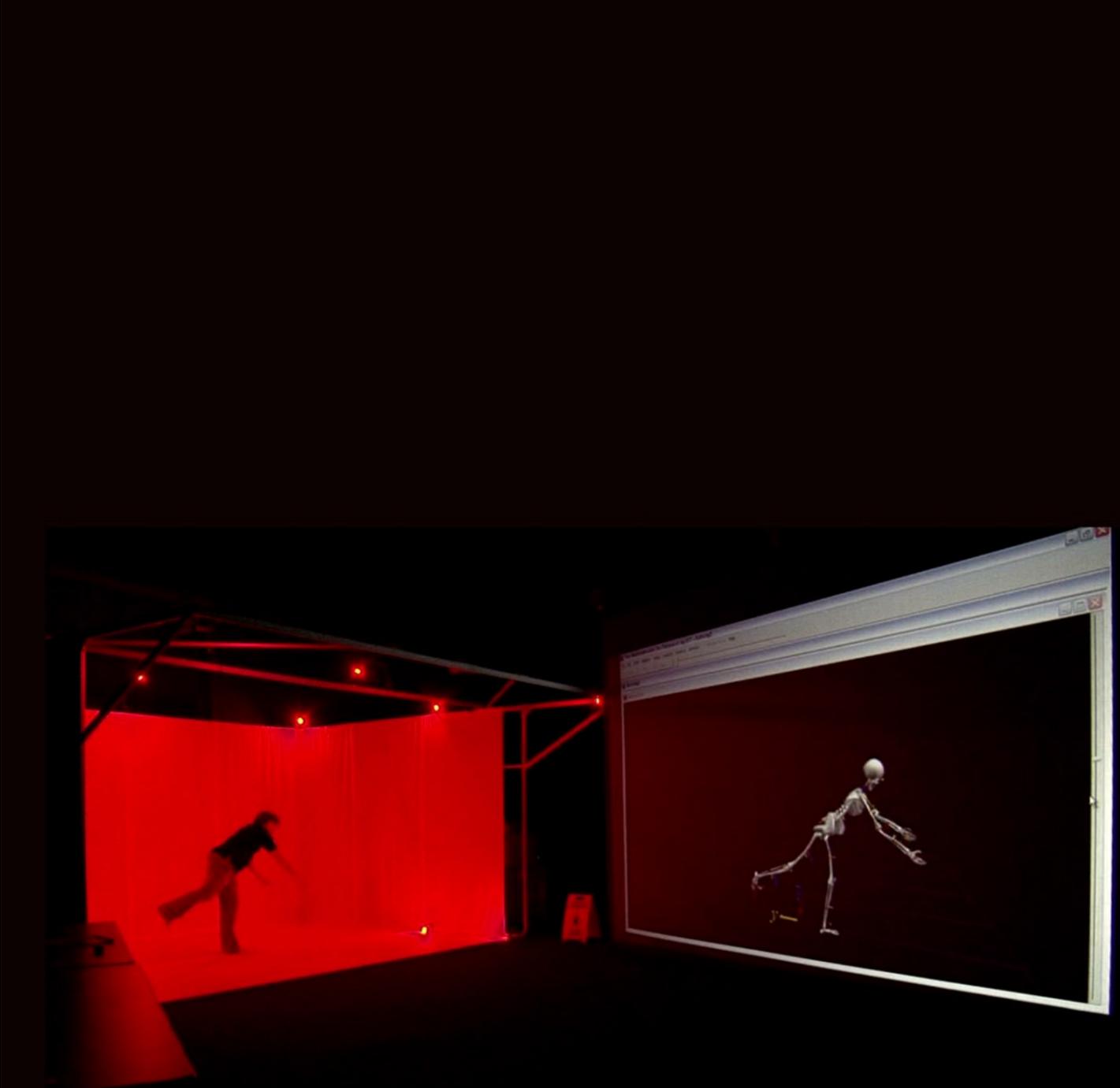
The 2,000 or more languages, that surprised Brooks, were mostly the product of that dominating imperative, and only rarely made things better overall.

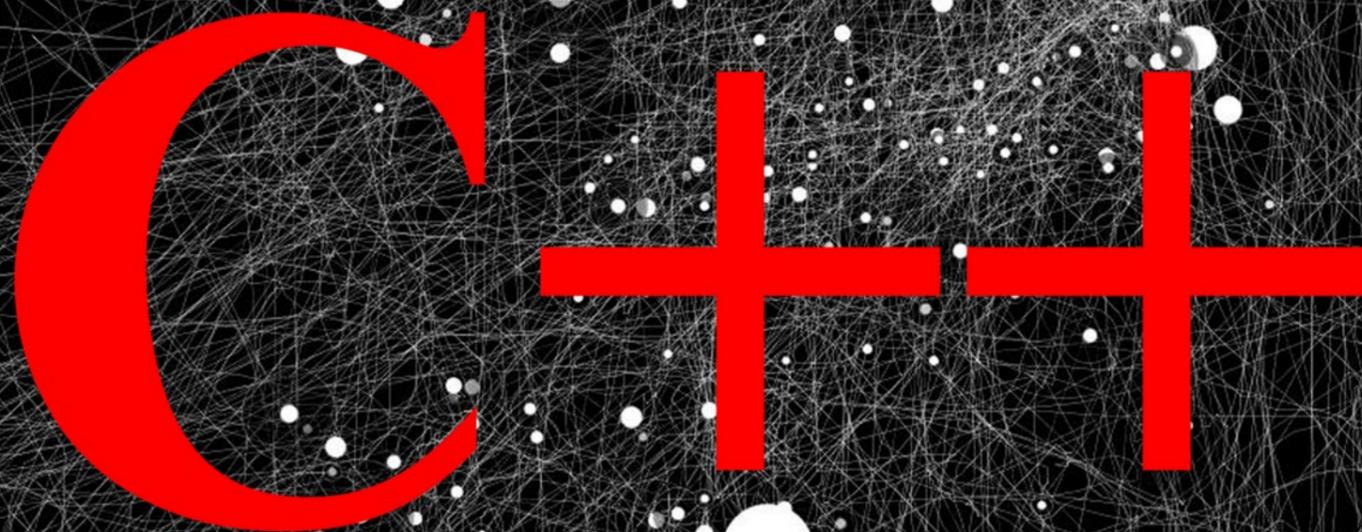
Indeed, as a long-time writer of computer software in many languages, I can confirm that even the earliest products, like Algol and Fortran, were more than capable of doing literally anything. And, most importantly, were much easier to debug – to reveal and correct the errors made.

A simple case will prove my point: A Department of Computing in an English University (many years ago) was developing software to reveal the electron "orbits" involved in molecular combinations, and used an enormous computer, and all the latest techniques and software to do it. The final success was lauded in the Journals, and the Department accrued a great deal of credit from their achievement. Yet, a colleague of mine, whose degree had been in Chemistry, but who I had recruited into Computing, used Algol and a small obsolete, mainframe computer, and achieved exactly the same thing, by himself, and in a very short period of time. He was, of course, in possession of all the understanding, as a chemist, to provide everything that was essential from that side, and his new-found skills in Programming could be effectively marshalled to very efficiently produce what was needed. The real achievement of the University experts was in their own discipline – Computing, and NOT in the discipline their software was designed to aid.

Our culture, and particularly that connected with computers is most definitely Technology-led, and that doesn't only mean in the speed or miniaturisation of the kit involved, but also in the techniques and methods developed in software. It is very inward-looking. The nearest equivalent to this constituency must be that of the mathematicians, who also develop their discipline in its own terms to the utmost, so that the level of abstraction takes them inexorably into Singularities, multiple dimensions, not to mention Parallel Universes.

Why? It is because their discipline allows them to do it. Yet, they firmly believe that the relations and formulae, that are their bread and butter, will enable all problems, in all areas of Reality, to be solved. But, by the very nature and methods of their discipline, they only work within the World of Pure Form alone, and never in concrete Reality as such. How can they solve anything that is quite firmly within the all-embracing world of concrete Reality?





C++ and the Philosophy of Mathematics

1: Aspects of C++

1.1 *Overloading - Overt Generality & Hidden Detail*

The most interesting feature about C++ is the facility called function overloading. This, on the face of it, seems only to allow quite different functions to have the same name, but, there is a great deal more to it than this.

First of all, this isn't as confusing as it might sound, because WHICH function is used at any given point in the program, is dependant upon the context in which you call it so that (if you do your work correctly) the appropriate one is called when needed.

What then is the advantage of using the same name? Why shouldn't we use different, or modified, names for each different case? Well, yes and no! If all functions are named differently the situation can become very confusing. A collection of functions covering the same sort of processes - a family likeness set of functions, if you like - would each have to have a different name, showing, on the one hand the family likeness, and on the other, each subtle difference. Any naming convention to deliver all this information, would need to be complicated, and STILL could lead to confusion.

Take the situation of a complicated package with a series of different, but related, treatments of certain data. If all alternative treatments have different versions of the same name, you actually have to remember ALL the detail to be sure you are picking the right one at any time.

The advantage of having the same name is that you don't have to sort through a large number of alternatives to call the correct version. After all you MUST be setting up in order to process a PARTICULAR case at any one time in the programming, and this CONTEXT structure can be made to CHOOSE the appropriate version. The code that deals with each case then becomes IDENTICAL in the CALLING part, and only the appropriate conditional tests need be concentrated upon.

This turns out to be a much more organised way of dealing with such situations, NOT ONLY for a single programmer, but most especially when a team of programmers are producing the package, and different individuals are dealing with the various alternative cases.

In addition, this overloading allows you to think at a higher level = the details are dealt with at a lower level - WITHIN the individual function, while at the higher level they are covered by a single function name - A SINGLE CONCEPT!

1.2 *Operator Overloading*

Similarly, certain OPERATORS can be overloaded. The clearest example is with such operators as +. This can be made to mean numerical addition, concatenation of strings, or even complex procedures such as matrix, or vector addition.

What happens is that you define your special operations when you deal with them in detail, when you are addressing the particular requirements of that special case.

Now the above description, though correct, does not do justice to the concept of overloading. What makes the whole thing so powerful is the feature of C++ called OBJECT ORIENTATED PROGRAMMING. More will be said about this elsewhere, but for the purposes of describing overloading let us briefly explain the functionality here and now.

Object Orientated Programming arose out of the feature in C called Structures (structs), where sets of variables were grouped together under a single name. Because of this whole structures of information could be dealt within a clear and simple way. What became clear is that such structures ALWAYS required their own functions to process them, and a very powerful advantage could be achieved by GROUPING the functions AND the variables TOGETHER in one definition. This grouping was called a CLASS.

It should be coming clear why this was so important. Within a particular class, with its own data structures and functions, it would be nice to DEFINE also its own OPERATORS to manipulate these structures. Adding WHOLE structures together could be achieved by OVERLOADING the addition sign (+), to do this.

1.3 *Top Down & Bottom Up*

So now, when you are looking at the structure in a TOP - DOWN way - an overall way - a place holder single NAME, or a placeholder single OPERATOR will cover different functionalities, while providing a conceptual level for what is being done.

At first I wasn't particularly enamoured of this feature, but I believe that it can be helpful to programmers, perhaps the majority when writing code.

I have a long standing hobby horse about programming which is related to this discussion. It is about the strategies of top-down and bottom-up in writing programs.

It has been the received wisdom for a long time now that programming should be designed top-down and anything that facilitates this is to be recommended. The above discussion on overloading definitely concurs with this principle. It facilitates a top-down approach.

In spite of differences that will be encountered in a programming task when the detail is addressed, at a higher (and EARLIER) level of work, PLACEHOLDER (generalised) function names will help the programmer and avoid the “can’t see the wood for the trees” problem that can certainly come from a purely bottom-up approach.

This is all true, and particularly appropriate in (for example) commercial programming environments, when the tasks, are mostly those that HAVE BEEN DONE BEFORE. Maybe its a database, or a payroll, or a financial balance sheet, or whatever. The DETAIL may be different, new facilities may be being brought in, BUT the general approach - the concepts involved - is going to be the same, and a generalised top-down structure in the coding will enable large chunks of previous code to be re-employed and the task becomes one of fitting these ready mades together in a top down way, with a sub task of “differences in the detail” - the LOWER LEVEL FUNCTIONS in any NEW features that are being added, and will fit into the top down structure.

In addition, the top-down approach enables large teams of programmers to work together, or even NEW groups or individuals to re-use existing code to MAXIMISE its use in “infinite variations on a theme”.

Also, as long as the code delivers the functionality required, original uses (or should I say REMIXES) of the code can be developed. The limitations are that

1. you accept the code as it is
2. you accept the implied top-down forms

The best example is the Windows environment, and particularly the multimedia MCI drivers with their standard interfaces. These allow software writers to harness the multimedia functionality directly into their code.

ASIDE: A discussion of this really needs to investigate whether these are used as much as they should be. My own experience does not confirm that this use is very common - though I may be wrong . Maybe ActiveX technology is being used in this way for games - though if this is the case I can see NO innovative uses being developed in my area of research, multimedia and Dance.

Hopefully, I will be able to add a great deal more to this part of the discussion after my current investigations into

C++ and VFW, XTRAS, Xing, ReallImage and Director Plug-Ins. If I manage to write the code I require and if I get a good handle on Plug-Ins, (Media Mogul & others) then I will be able to fill out these points much more fully.

Now, my position is different to the consensus in ONE very important area. That is when the programming being undertaken is in completely *original* work - research work - never before attempted or even thought of.

Areas where new techniques and inventions are unavoidable, where entirely new ways of dealing with things, new and very different demands made on functional facilities, and where new aggregations of these sets of functions are required, CANNOT be tackled in a top-down way, at least in the crucial and formative stages of development. Top-Down methods ASSUME that you ALREADY HAVE the solution in a general form.

In the special circumstances of real innovation that I have outlined, certain work HAS to be approached from a bottom-up direction. Detailed problems with NO, as yet, clearly defined solution, have to be tackled FIRST.

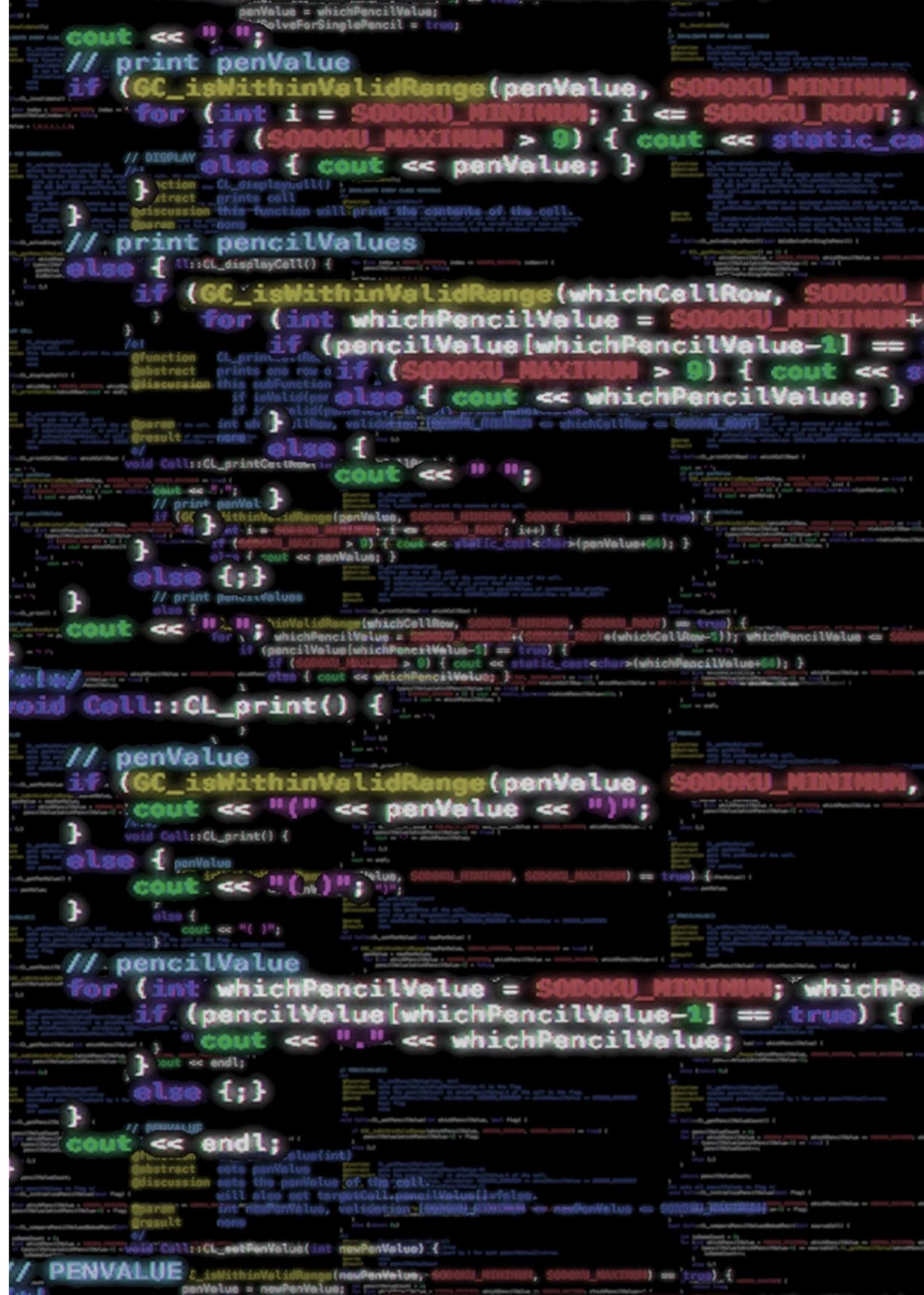
In my experience, innovatory software is generally a see-saw approach oscillating between bottom-up and top-down approaches, AND, what is very important here is that the FORM of the top-down approach is ALSO INNOVATORY, and could not have been commenced BEFORE the inventions and techniques had been developed during the bottom-up investigations.

If this is so, then a purely top-down approach COULD NOT deliver truly innovatory software solutions to problems, but would be essentially RETROSPECTIVE. You could say that the approach is ideal for mediocrities doing rehashes of well established tasks, using as much ready made material as possible. Perhaps that is a bit harsh. Though, in a very important sense, it is true. In my own work, which is innovatory, I seem to get little help from facilities, libraries and even packages & operating system facilities.

I would even say that if top-down methods are insisted upon, it is certain to be CONSERVATIVE in the top-down approaches used. That is NEW top down approaches will meet great resistance and incompatibility from the established methods.

As I have said before, if it is bottom-up solutions that lead to NEW problems, and these need to be solved by INNOVATION in the form of top-down methods. then the insistence on starting from (established) top-down methods, EXCLUDES this innovatory process.

Once a new software technique has been made available - the vast majority of use of it is top-down, and the form of the top-down use is usually “cast-in-concrete” within



the package or the tool which delivers the technique. All products have the stamp of the tools used.

This is because it is usually so difficult to BEND the functionality to original aims. You do what the package let's you do! Your top-down forms are constrained to those available in the package - if no alternative uses were realised in the course of the programming of the package, then you, the user, wont be able to do them, even though it should be possible, given what the original authors achieve - that is the lower end functionalities must be there, BUT you cant get your hands on them. In addition, very useful lower end functionality which could very easily have been provided, if your new use had been considered, DON'T EXIST and are not really feasible, for immediate implementation, IF EVER!

1.4 The Connection with Abstraction.

Now this aside is taking one important aspect of the philosophy of C++, but there are also many connections and similarities (and differences) with the philosophy of mathematics, about which I have been writing elsewhere in these notes. This is the role of Abstraction, and will be taken up later.

2: C++ and Mathematics

Let us attempt to relate the overloading of functions, operations etc. in C++ with the switching of formal elements between levels and context that occurs in mathematics.

The simplest place to start is probably in the overloading of operators. Elsewhere in my notes I have discussed at length the regular extension of the concept of number, to allow the extension of the whole system of operators and number functions to other areas with "similar" formal relations. In effect the overloading of operators in C++ is directly comparable with this. Such operators as + can be extended to apply to vectors, matrices or what have you in BOTH.

Also in both, to be able to immediately be in a position to think about the whole set of operators available in one area, but perhaps applicable in another, is a very powerful conceptual approach. and (in the case of C++) if you have made damn sure that the detail of a new use for common or garden operators has been thought through and implemented unambiguously (in when it should be applied), then you are free to "lump them together" in thinking at a higher level (The famous top-down approach).

There are similarities again with mathematics in that mathematicians do not "willy nilly" use operators in a new situation. They also define new rules and procedures for their "overloaded" extensions to number theory.

It is clearly a formal question! A question of levels. Mathematicians (and perhaps all of us), prefer to make new areas "fit in" with their already thoroughly studied and greatly used techniques. It is easier to add extra rules, constraints and exceptions to the already established, than to insist that, because of differences, the new area must be partitioned off. It is also, I may say, a SOUNDER approach, because you are trusting to the formal architecture ALREADY established, rather than assuming that things must be different and contradictory.

History has proved the efficacy of the mathematicians approach, though the always reappearing crises of mathematics, are also based on this assumption.

This may sound contradictory, but it is also true!

ASIDE: It is clear that as the procedures in C++ that we are discussing here were being developed and implemented, ambiguities did appear and things did go wrong. Corrective procedures and special representations were then developed, which (surprisingly) tended rather to be against the general philosophy, in that CONTEXT labels have become more and more necessary to remove ambiguity (e.g. Date:: etc. class labels).

In addition, a proliferation of seemingly redundant symbols have become necessary to tighten up the meaning of certain constructs, for example:

date()

redundant parentheses in functions

*T &array <T> :: operator [] (int index)
delete [] storage*

opaque sequences of symbols

To an experienced programmer used to the old procedural languages, the appearance of C++ coding was at first meaningless.

In a real sense, it is very opaque! Perhaps the approach (- a computer science approach, as distinct from the user orientated approach) has led to a minimalist, very spare and “symbol-flagged” form , where sequences of symbols, in a given order and context, identify situations. It is the reverse of the old “English-like” languages (COBOL etc.) - [though I must say the interminable forms of COBOL used to bore me to death]. It is much more akin to low level languages in this respect, though, of course, its Object Orientated Approach and hierarchical functions make it much less like assemblers and the like!

For the Initiated Only!

One particular aspect of C++ that I find very interesting, is its inaccessibility. No-one, who isn’t a very competent and experienced programmer, would make head nor tail of it.

It seems to take PRIDE in its inaccessibility. It is almost an initiation! A separation of US from THEM. The real computer scientists from the amateurs. The experts from the users.

If anyone had decided to invent a language that would be impenetrable - a secret language for initiates only - then it would be very like C++.

You have to “learn the rules”, “know the ropes” and some of them are directly opposed in form to the more usual methods in the procedural languages, and also it is very “codey”.

Example: The use of begin and end in the earlier languages has been carried over to [], and the normal uses of parentheses (), have been shared between {} braces,[] brackets, <> pairs of inequality signs, and () parentheses themselves, and some of the sequences of these are quite off-putting (see examples above).

JS (1998)

Painting By Numbers Jigsaw Programming

Modern Authoring Systems – A Boon to Programmers?

Having just wasted a vast amount of time struggling with ActionScript programming in Flash MX, I couldn’t help but wonder why? Well, I am sure that computer “nerds” of all types will smilingly inform me that it is because I am either too “thick”, too-long-in-the-tooth, or simply stuck in my old-fashioned ways to cope with the latest state-of-the-art tools. But the prejudices of youth tell us nothing. If anything they tell us more about the judges than the judged. My own credentials can be assessed by the fact that I have been forced to learn new authoring systems for each of my last four multimedia packages. Packages which have, each in turn, been generally agreed to be the industry-leading resources in my area of interactive media for the teaching of Dance. *NOTE: My first multimedia project won an award at the BIVA event in Brighton in 1989.*

The trouble is that the current generation of authoring “systems” are supposed to make the job easier. The crucial question must be, “Easier for whom?” Personally, I have a great deal of trouble with software, that expects me to “work purely from examples”, or even take “ready-made” chunks of code “on trust”. I need to understand “Why?” to be able to realise what might be possible with the current tool. Patchworking together ready-mades is no good to me. The truth is that such methods are NOT intended for the professional, at all, but for the “amateur”. If the user is merely “remixing” previously solved tasks into some conceptually well-trying undertaking, and, considers the creation of original and creative code to be not-for-them - then , this sort of tool is ideal. If, on the other hand, the user’s purpose is to invent, wholly new software facilities, then such tools are a major disadvantage, because the-building-is-determined-by-the-bricks.

Do What I Do

No real explanations are given along with such tools! No structured teaching is involved. (I think I will have to repeat that – no structured teaching is ever involved with such systems!). To those who insist that I have got this wrong, and that lots of help is instantly available, let me assure them that being led by the nose through individual examples is NOT structured teaching. It is “do-what-I-do” training. It almost never leads to a real grasp of what is going on. It involves no attempt at explanation. It, at best, allows dedicated “nerds” to pack their brains, (remembering rather than understanding), with a multitude of techniques that work. – “Who needs to know why?”

Problem-led Software Facilities? – “Not Today Thank You!”

My proof of this is that whenever I approach such people with a new problem, they invariably insist “You shouldn’t be trying to do that. You should be doing this!” They find such problem-led creative computing to be a pain in the ass. They get more than a little upset if you insist upon addressing the problem defined OUTSIDE OF their preferred computing field. The only discipline-led projects they are interested in are computer-situated projects. I have witnessed innumerable so-called “joint” projects with all sorts of intelligent partners with fascinating requirements, which have rapidly turned into vehicles for the computer buffs to display their own totally dis-embodied computer skills, and which have totally failed to address the actual real-world problems which were the impetus for the partnership in the first place.

Jigsaw Programming and Painting by Numbers

Let us dig a little deeper into these “revolutionary” tools. The most significant facility in these scripting systems is undoubtedly “Object Orientation”. This facility allows new users to use previously written, second-hand scripts, and simply use them in a completed and handed down form. Collisions of names used (for variables, for example) doesn’t matter (we are told), because such tools also allow so-called “overloading” and the names are all local to the individual objects used, and , as long as we slot in the definitions – as given, and fill in the “blanks” of the given template, all should be well. Understanding these pre-formed building blocks doesn’t seem to be a necessary part of the process. “Painting-by-numbers” programming using given chunks of jigsaw-code seems to be the rule.

Of course, such methods are fine for amateurs or hacks, doing what has been done thousands of times before – where only the names need to be changed (to protect the innocent?) For, hack programmers simply doing yet another standard application will rejoice at such facilities. Merely re-filling the parameters with appropriate values or variables is sufficient. The trouble arises when the programmer needs to invent – to do something new – to develop new methods and techniques, and stretch the authoring system up to (and beyond) the limits laid-into the bricks to be used. Such work is generally scuppered by systems such as the type I have been describing. They do what they do. And I suppose that is what the vast majority of users want them to do.

Everyday Tools for Everyday Tasks

These are certainly NOT development systems for creating original techniques. If you want to do a simple database, a catalogue, or a standard accounting job, these ready-mades are fine. But, I am trying to write state-of-the-art techniques in handling quality, full-screen video tailor-made for the teaching of Dance. Such techniques do NOT yet exist. Predicated, as they are, on full-screen, full-motion video, with detailed and accurate controls, the possibility of adding value, and of extracting qualities from the source video for revealing display in new and informative ways, all these methods have to first be invented, and then implemented.

Most of my career has involved resorting to levels in programming—when particular primitives are not available, you write them yourself at some lower level (assembly language or whatever) is appropriate. Alternatively, you search around for the facilities you require in 3rd party software, and move around between the various tools to bring together the required functionality. Sometimes, you acquire 3rd party Plug-Ins, to enhance your main software tool, and bring to it the required additional features.

New Media, New Methods

Before the new amalgam of video, computing and pedagogy, it was impossible (at any reasonable cost) to deliver the sort of added-value resources that are possible now. Over the last few years many new innovations have been conceived and delivered by the author in this area. Perhaps the most effective was Historical Equal Interval Still Sequences (called HEISS), which addressed the contradiction between Dynamism and Context in studying movement, and presented sequences of stills alongside video materials to aid analysis. Subsequent developments from this such as synchronised HEISS pathways were also superimposed on top of the original video as either static or even animated & synchronised overlays. Many other techniques were developed to allow effective interaction with the resources, without restricting the full-screen requirement for the underlying video. They involved a variety of Head Up Displays (HUD) for a wide variety of options and effects.[HUD involve overlaid transparent data and interactive buttons.]

All of these are original. You do not find these as ready-made facilities in the scripting systems that I am talking about here. You HAVE to write them yourself, and the “helping-hand”, built-in methods of the scripting language seem only to trip-up the serious programmer. After a great deal of work, I have sorted out many of the “given” (yet unexplained) templates, and have developed many effective solutions to Dance-and -pedagogy problems that were constantly presenting themselves, but the amount of work was enormous. Why?”

Inadequate Teaching – Training is Retrospective

Because to create such facilities, you HAVE to understand what is going on, so that you can effectively USE what is available. Lack of any real explanations by the language authors, caused erroneous assumptions on my part, and led to a great deal of wasted time. Most of the time is spent first understanding every line of code given as examples, and then re-casting the elements of legal code into something QUITE different. The debugging techniques available are crude and interminable, and surprisingly old fashioned. In today’s environment of power and speed in computers, you would think that a second screen, with real time updates, slow motion running etc. would be standard procedure by now. But then that assumes that the tool is intended for the professional programmer. It is not! It is intended for the amateur, or the hack. And, you don’t spend enormous amounts of time educating amateurs who only want to play, or hacks delivering the same stock solutions, do you?

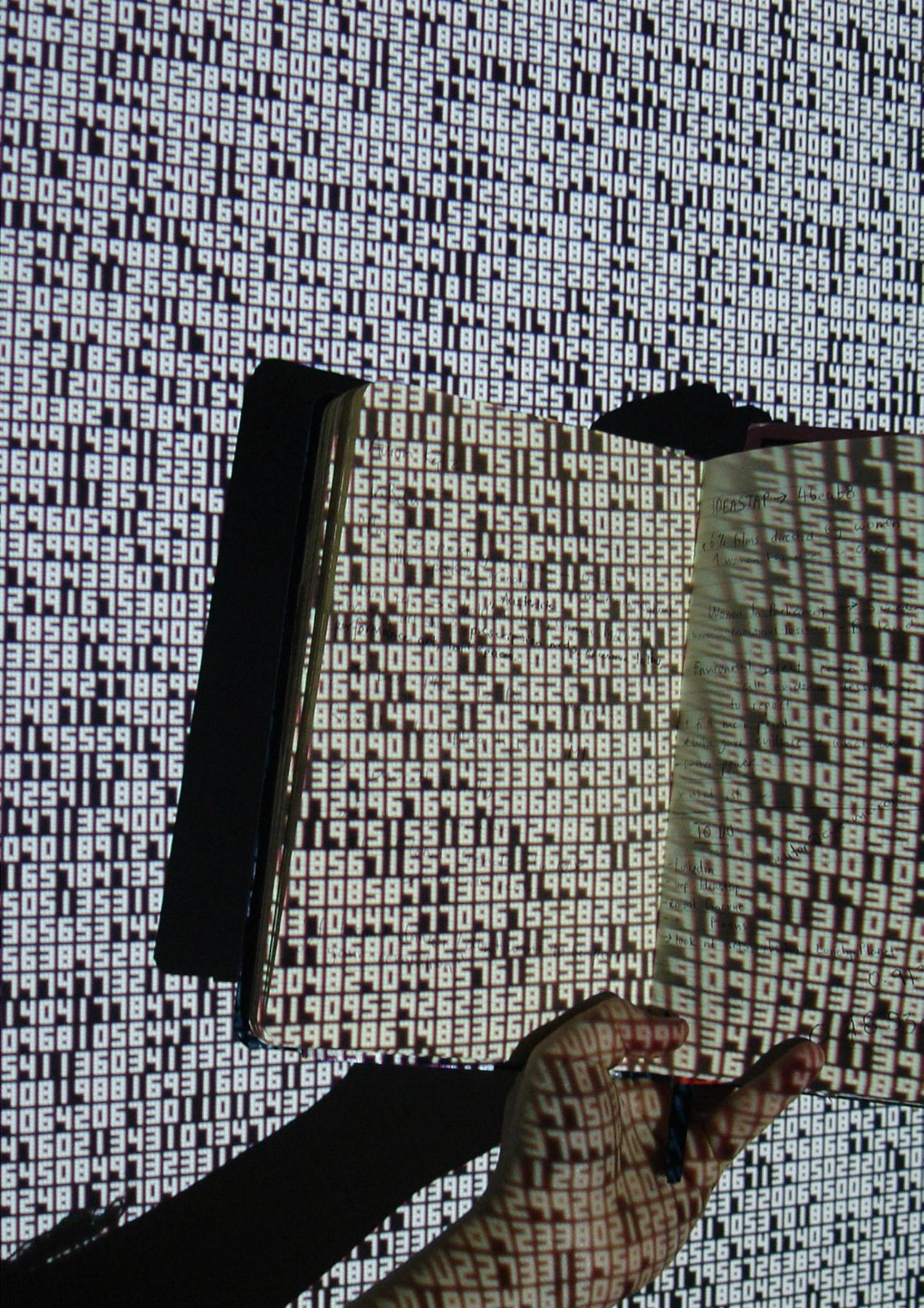
Inadequate Programmer’s Aids

More often than I care to think about, I have to note things on paper, in order to have crucial values available elsewhere in the code I am developing. For example, I spent many hours constructing essential tables that relate all the parameters, and show them simultaneously, and in clear relationship to one another. Parameter presentation tables would seem essential facilities to me.

Causes, Implications & Effects

If my criticisms are true, what are the implications for the development of new techniques in information technology? In contrast to the consensus view on this question, I maintain that the consequence of this situation is not unbridled change and innovation. On the contrary, I would insist that the development of these tools is essentially conservative and retrospective.

Why? It is because they let more people do old things. They probably let them do these established techniques more quickly too. But, the incessant thirst for innovation and the “latest thing” (which is now a firmly established part of the technological myth) is considered to be adequately fed and entirely satisfied by speed and capacity. It isn’t radically new techniques that are the motive force for sales, but faster, more capacious devices – greater detail and image-resolution etc. etc. In fact, really new ideas find it hard to get a hearing. What gets taken up are contributions that “fit in” with the consensus ethos, whereas new ideas are too often considered to be either too challenging or even too undermining. Now real progress does still happen, but it isn’t in the mainstream. The really important developments take place elsewhere. Usually, these are produced by unknown players in small companies, and the process of integration of their work into mainstream products is painfully slow.





Flash

The developments almost always take place in spite of the prevailing system NOT because of it. Generally, they get taken on when the big players begin to feel threatened by their much smaller competitors, who are then emasculated by take-over or head-hunted to acquire their star developers.

Backwards Development

If you find all this hard to believe, you will be even more astonished by the following statement:

In some important areas the flow of facilities seems to be backwards.

Important facilities in my field grow less and less instead of more and more. While products get faster and more capacious – much trumpeted – they simultaneously lose vital facilities. In Dance Multimedia, (which, by the way, requires very similar facilities to the whole range of sports studies), the authoring technologies, and their associated software tools have progressively deteriorated quite significantly since the 1980s. When I started in the field, I used Philips Laser Disc technology controlled by a cheap “school-type” computer, and produced a pedagogic pack that was good enough to win a national award in 1989. Since then the facilities I took for granted (using the above set up) were gradually cut down, in subsequent media technologies and software, so that most of the innovative techniques that I had developed were less and less possible. The declining sequence was:

Laser DisC > CDi > CD rom

Appropriate Tools by Accident

And, it is only in the last few months that the facilities that I used in 1989 are becoming available again – 13 years of having to compromise instead of rocketing ahead!

And these new facilities are available almost by accident. It turns out that the requirements of the “latest thing” – the Internet – has driven the development of “new” tools. The Internet is simply too slow! This godsend has failed to deliver the promised goods, and short-cut solutions had to be found. In addition, to at least give the illusion of interaction and multi-parallel processes, it became necessary to allow very “busy” screens – full of lots of little independent animations doing their things simultaneously on screen direct from the Internet. Now even these don’t really work if all the resources are situated at the other end of the Internet link. So, the whole philosophy of automatic downloading of resources to the user’s own computer was developed, not to mention the added value of “streaming” – that is playing while still downloading. It was also necessary to accelerate the speeds of user’s own Hard Disks to co-operate in these new activities.

These imperatives may have helped the burgeoning internet industry, but they have also returned to multimedia authors, such as myself, the wherewithal to once again produce quality pedagogical video based materials direct from CD rom (or from a local Hard Disk) using applications such as Director and Flash.

I use their tailor-made tools to give me the facilities I have been missing since 1989. Our dance products using these facilities are the best in the world, and we have finally re-obtained the tools that are necessary in this field. Interestingly, I have presented many papers on what I considered essential for quality, pedagogic, multimedia resources, AND the tools to produce them. I even wrote extensive pieces, which I sent to Microsoft in Seattle – to NO avail! Nobody was interested. It is ironic that with everybody looking the other way, they have by accident returned to me the facilities I have been demanding for over a decade.

Dumbing Down?

The Consequencies of OOPS systems

If what I say is true about the current types of OOPS (Object Orientated Programming) software systems, where most users uncritically use blocks of code without any real measure of understanding, then there are significant consequences for the Information Technology Industry (and , of course, its users – us).

First, we must assume that ever higher hierarchies of “levels” of use will be built up, with each level knowing little or nothing about the real content of the level below, and of course, almost nothing about even deeper buried levels. Now, before a mass of people jump up and protest that you don’t have to be informed about the innards of a radio, or a washing machine in order to use them, may I ask a simple question? Can I ask if they also feel happy with a situation where the man called in to fix such a machine also knows little or nothing about the workings, and suggests an immediate replacement? If you think that such a scenario is far from the situation prevailing at present, then read on!

IMPORTANT NOTE: One aspect of all this that is never addressed, is the foundations upon which it is all constructed. Not only Programming, but Mathematics and Formal Logic all depend upon the Principle of Plurality for the whole standpoint and methodologies involved. This reveals itself in Analysis, and Reductionism, which can only be legitimate if extracted elements and relations are entirely independent of context: we say that they are separable as-they-are, and are not modified by context.

Now the above points made about the multiple uses of blocks of code, is clearly related to these assumptions. Are they too considered to be both separable and independent of context? The alternative holist standpoint insists that they are NOT separable, but are changed by context. And all causalities are therefore not all bottom-up, but also top-down and even side-to-side.

Minimal Understanding with Maximal Reach?

To return to programming – if my representation is true then enormous amounts of code will become involved in all implementations, in which NO individual author goes beyond his or her own level, and most of the basic code is hidden and never really analysed or understood. Such a situation has a momentum of its own. Situations have occurred before in the development of Information Technology with important consequences – from do-everything compilers that ground to a halt under their own size, to “universal” solutions packed with inefficient code. Of course, you may say that this could be true, but who cares? At the rate of development of computer hardware at the present time, such inefficiencies can easily be compensated for by increased speed and capacity [You can’t beat faith, can you?] But I’m afraid that no growth process can be relied upon to give such regular rates of improvement forever. Nor can layer upon layer of ever increasing (and largely unknown) complexity be counted upon to generate no problems of reliability – to be totally sound, and unlikely to fail.

The Flight from Explanation

We must also couple this mounting bulk of hidden code with the increasing flight from real explanation and teaching in this and every other field at the present time. The consensus is now that teaching is old hat. (I recently attended a conference where a leading member of a government funded organisation given the task of promoting Teaching and Learning in the Performing Arts, who vigorously upbraided me for asking for the discussion to get round to the title of the current session – Teaching). The future, we are told, will be carried forward with “learning” systems, for example – the Internet, or in help files on disc. This position has rapidly eroded to become the ancient method of “Do-as-I-do”, where the writing of “training materials” simply leads those trying to understand an area covered by a piece of software, being led through step-by-step sequences that give various standard results.

From Programmed Learning to a Compendium of Particular Recipes

As Programmed Learning proved way back in the 1960s, such methods cannot cover all eventualities. I was working in this field then, and it became obvious that the development of such comprehensive materials – attempting to cover every possible case, rapidly became uneconomic.

In fact, the whole exercise was abandoned as impossible. A team would take years developing materials covering every conceivable eventuality, and at the end of an interminable process, the result could be very easily criticised as partial, or biased, or using limiting assumptions. What is remarkable, and dangerous, about the emerging second “era” of Programmed Learning, is that the new generation of “trainers” don’t even attempt to address problems generally. They simply proffer single sequences of actions that address SINGLE problems, and give single results. Such particular “training” sequences cannot be criticised in the same way as the materials produced in the first generation of PL in the 1960s, because each sequence only attempts a single particular outcome. No attempt at deriving concepts, or establishing an overall view is attempted. Anyone should be able to follow an individual series of simple instructions, so everyone should be able to achieve the single objective! For other outcomes, the learner must find other training sequences, and the whole approach can only lead to thousands of particular examples. The job of the learner becomes one of finding these (hence the Internet) and trying them out. If one works, it can be added to one’s armoury of techniques. The role of the learner is to amass innumerable, black-box techniques with NO real explanations or meaningful commentaries.

Such sequences of instructions are so devoid of significant meaning that if the user doesn’t constantly use them, they are forgotten, as they don’t fit into a matrix of understanding, with cross-connections, and conceptual hooks. They are not memorable. They are RECIPES!

The Deification of Arbitrary Memory

The amazing thing is that “nerds”, who incessantly use these “solutions” every day, soon are regarded as “experts”. Teenage boys, who spend vast amounts of the day (and the night) “learning” sequence after sequence, are rapidly seen as geniuses by the general public. Yet, whenever I approach such people with an intractable problem in my own researches (note that this invariably means that it is unlikely to have been turned into a sequence of actions for general use), I invariably get no help at all. In fact, such “experts” admonish me for posing the problem in the first place. They don’t solve problems: they remember processes! Their “expertise” is in fact retrospective and conservative.

Debugging & Help

I have recently tried to get up to speed on ActionScript (the scripting language in Flash), (as mentioned earlier) and quickly found that the manuals given with the software were inadequate to my needs. They were catalogues of innumerable definitions and processes, and did not allow any real comprehension of the system as a whole. No teaching was involved, only the provision of masses of separate details. I also found the Help files accessible either

via the software, or on disc, or even over the internet to be insufficient. In the end I was forced to buy FOUR, quite thick books to supplement the Manuals and Help Files, But, you have guessed it, they were ALL sequences of actions to merely achieve the most common or garden results. They were for amateurs or hacks doing ordinary things. Real creative computing cannot be built out of an infinite number of particular cases, but only out of *conceptual* understanding. The number of things that are done with such ancillary materials get narrower and narrower – not least because long before any sort of comprehensive mass of techniques have been generated, the software is superceded. A new version, with new tricks is released, and the whole process starts again. So, it becomes clear that there will only be time to deal with the commonest problems in providing appropriate “training”. I have a large library of computer books that were essential for a short time, and are now totally useless!

Extrapolation to Repair!

To look at the situation from a different angle, let us consider hardware! I have a series of computers dedicated to different purposes, and regularly in need of repair support. “Repair your computer?” “No, we don’t do that!”, I am told. It seems that any process involving testing and diagnosis is not only time consuming, but also likely to fail! All such processes are rapidly truncated with the conclusion – “This Disk (or power supply, or motherboard) is finished! You will have to get a new one”. It seems that hardware put together in developing countries can now be obtained so cheaply, that a quick profit is made by simply trashing the old part, and replacing it with a new, cheap import. If, by any chance, such a procedure leaves you without crucial data, then you are forced to agree to “recovery”. This is provided by a very small number of companies, though an inverted pyramid of “middle-men” has grown up to take their cut, and pass the job on. Recently I had to recover vital data lost on a broken drive. The “recovery process” involved my paying for a brand new identical drive, which was then used to replace everything but the data-containing disc itself, and allow extraction of the required content. Note the method! No detailed and skilled testing and diagnosis was involved, with a single or small number of replacement parts bringing the disc back up to a fully working state. No, that would be too time consuming, and involve too little profit. A whole new drive was “necessary”, and the recovery process was simple and quick! The cost of recovery amounted to enough to buy two complete, and brand new computers. The process involved paying a firm in the USA, though I had contracted one in London to do it (which turned out to be just an office with a couple of employees). The point I suppose I am making here is that the same philosophy pervades all sides of Information Technology these days. Does my experience with the defunct drive demonstrate this? Well, if we extrapolate the “sequence-of-actions” methodology to repair, I think it does. That is, if real understanding is

replaced by recipes, then the shortest (and most lucrative) sequence must be “chuck-it and install a new one”. I’m afraid any hope that a philosophy akin to classic car restoration might occur is impossible here. No one wants to restore an ailing computer to its old fashioned, snail-paced perfection. I get monthly missives from a variety of sources invariably informing me of significant cuts in hardware prices.

Aside: A supporting anecdote is worth inserting here. In 1989 I authored a multimedia product for teaching Dance using a Philips Laser Disc system, and a BBC B microcomputer. The system handled full screen, full motion (25 frames a second) video, with finger tip control of pause, play, slow motion, step forwards and backwards, and immediate random access to any part of the video sequence. It did the job required perfectly. YET, within a short period the Laser Disc technology was dumped, and replaced with CDi (supposed to be better, but is wasn’t!) The computer was quickly superceded with faster cheaper models, BUT most of the features that made our original system, and National Award winner, were lost.- YES, LOST! The new alternatives did not provide them. New projects had to compromise – they were simply not as good as the original, because the essential functionality required in studying detailed and expressive movement had been excluded. In fact, I have had to wait 13 years until the “requirements of the Internet forced the industry to return the functionality that I required, though it has to be said that doing that was not their intention. Other different motives had forced the provision. The gap was from 1989 “The Dance Disc” until 2003 “Motifs for s Solo Dancer” without access to essential video multimedia facilities (such as cheap systems for dynamic/animated overlays).

JS (2003)

Too Many Notes...

After some 10 years of writing, and the last 6 of those full time, I have found it unavoidable to have to address a whole extended raft of areas, which not only relate with one another, but also actually mutually determine one another, to a remarkable degree.

To be a pure specialist makes the solving of many problems virtually impossible! And the reasons are obvious! Strict limitation to fairly narrow areas of study, and most particularly when what is actually studied is in a severely controlled and constrained area too, always and inevitably simplifies the “seen” realities into purely abstracted forms. We don’t see unfettered Reality as it actually is, but a very special “purified” reflection of it. So, the tasks that therefore present themselves may well be much more easily tackled, but always at the cost of losing any really comprehensive principles. They are no longer available!

The imposition of both restricted local principles and contexts, along with such narrowly constrained Domains of study is always a debilitating result when it comes to both origins and process. We are effectively slowing down the action dramatically, and then studying a “still” of what we see! Consequently, we not only restrict our study to very limited areas, but also invariably assume a kind of stability in what we study.

And each specialist area also quickly accumulates its own coherent set of things studied and theories extracted: they are “true”, but only of the prepared and constrained area. And when specialists from several areas come together to crack an evidently cross-discipline problem, they invariably fail, because each side reformulates the problem in terms of their own limited set of ideas and methods, and the opposing purposes of the various groups do not produce anything that transcends these divisions. How could they? Indeed, if there is any movement it is merely to occupy the more peripheral areas of their own dependable realms, without in any way challenging the founding and determining principles on which they resolutely continue to stand. Indeed, in many such attempted collaborations, the emerging dominant discipline in the given context invariably takes over, and the objectives of the perceived junior partner are almost entirely lost.

Now, the reader may ask, with justice, on what evidence were these generalisations revealed to the writer of this paper, and the answer is perhaps surprising.

For an important decade I decided to dedicate myself and my acquired skills and understanding in Computing to my many research colleagues in other disciplines, who were clearly going to benefit from the aid that computer control could deliver to their many difficult areas of research.

It soon became crystal clear that if I was to really assist, I could not merely parachute-in with my panoply of general computer skills, and quickly solve their problems in my own discipline’s terms. On the contrary, the only contribution that I could make was if I were to immerse myself as deeply as I could into their discipline and their objectives. Only if I were directly contributing to what they considered vital would I be doing anything worthwhile. Certainly I was not a visiting master of unusual skills, directly applicable to all disciplines no matter how diverse. Instead, as I originally found myself, I was a bearer of many techniques looking for an appropriate application. In other words no use whatsoever!

So, in subordinating myself to the new disciplines aims, I soon realised that absolutely nothing appropriate yet existed for the problem presented. I could see possibilities, but it was clear that force fitting their problems into inappropriate vehicles would be worse than useless. I had, with my colleagues as directors, to begin to answer their needs using what I could already do. It was not a one-size-and-type-fits all situation by any means. I had to re-create known applications by tailor fitting them to their needs entirely. They did not have to learn my area: I had to learn theirs! And, with the systems I produced, they had to be able to use in their own well-established terms – but better and quicker! Indeed, my main task was to constantly suppress my own within-discipline delights (in my specialism) and instead what I knew and could use ONLY as a means of empowering my different discipline colleagues.

Slowly but surely, significant contributions began to emerge, and indeed joint papers with the other discipline experts, enhanced both our statuses within our individual areas. Soon, all collaborations started to produce these joint papers, and the institution began to attain a significant esteem world wide for its original (and very quick) contributions in a wide variety of areas.

At first the disciplines involved were not a million miles from my already wide range of specialisms. I was already a mathematician, a physicist, a biologist, a sculptor and a computer scientist, as well as having serious interests in other areas, so I did have a good start. But, nevertheless, the above principle of subordination to the requiring discipline experts applied in all the areas of collaboration, which soon included Mathematics, Taxonomy in Invertebrates, Mechanical Engineering Test Rigs with computer control & robotic measuring devices, Medical Treatment Techniques (radioactive anti cancer treatments), Optometrics, Eye Defect diagnosis (colour blindness) and even an original computerisation of a Gas-Liquid Chromatograph.



But the major breakthrough came in the most unlikely of all collaborations. In a new post, I looked around for similar collaborative research, and found the best possible application in Dance Education.

Most of my earlier efforts had involved Computers-in-Control, so instead of the usual areas of computer applications, where I concentrated upon wedding computers to complex kit, and thereby attaining all the merits of Computer Programming to such equipment, I had to consider a very different approach.

So in Dance, the obvious immediate demand was for perfect and flexible access to, and control of Dance video footage. My dance teacher collaborator and I began to criticise tape-based resources, both from these aspects and from the choices in what had been recorded. “Entertainment videos” were the only educational sources available, and they were frequently useless in conveying exactly what the dancers were doing. We therefore not only added immaculate control and manipulation via computer control of video discs (at first with Laser Disc, but later all succeeding delivery methods), but also added multiple cameras to capture otherwise unseen parts of a movement, and also to highlight details. We very early on synchronised alternative views of the same movement, so that they could be delivered in synch and simultaneously on screen for study. But, it soon became evident that we could also supply notation, both Benesh and Labanotation – on screen and synchronised to the action. But, we, in addition, began to divide all pieces into phases, sub-phases and even individual movements, all of which were identified by naming them (often with descriptive phrases). We then represented each subdivision by a rectangle (with its length determined by its duration), and delivered these as a mapping of the whole piece. They could each in turn light up in synchrony as the dance progressed, or could be used as a means of precise access to a required interlude. All of this and many others (with which I wont burden the reader with here), transformed the use of video footage in Dance Teaching, and the most important aspect has not even be described as yet.

The control of the video resources was immaculate. From the outset we allowed full speed, slow motion, forwards or backwards, frame-by-frame stepping through, local interlude review systems, looping around a fragment, and intuitive switches between viewing modes, where detail pieces, and overall views, could be seen simultaneously and in perfect synch. The concentration was on making the use of these facilities as easy and intuitive as possible, and to see a fine dance teacher’s using of these resources was always a joy to behold. We totally rejected the usual Programmed Learning techniques with all its many almost totally insurmountable difficulties, and instead delivered Resource Based Teaching Materials to be used in whatever way the teacher required.

In a test run for a GCSE exam on performing a given Test Piece, our separate (non examined) group did incomparably better than the usually equipped and normally taught participants. We were obviously working in the right direction.

But, returning to the main purpose of this paper, when considering the necessary revolution in scientific assumptions, principles and methods, the range of areas requiring total overhaul is both wide and deep, and no single discipline would ever deliver sufficient to enable a single comprehensive and totally coherent approach. It would have to be not only inter-disciplinary (writ very large), but even that could never be sufficient. For the gaps between the Sciences are clearly unbridgeable using current ideas and methods. The transition between disciplines, though man-devised, are even more a feature of the bases on which we continue to address Reality. They simply can never cope with creative, qualitative developments. And this means that what is crucial is not the usual type of revision as has occurred in the Sciences in the past. It will not be about transitions between such eternal categories: it will be about the emergence of the wholly New-creations, which cannot be logically derived from prior circumstances. The process of Real Emergence is a very different thing, and one that our misplaced assumptions and principles just cannot deal with.

So, perhaps surprisingly, the most important area will have to be in Philosophy, and in the crucial area of Form, we would have to primarily address significant Qualitative Change. Indeed, the almost never studied Emergences would certainly be by far the most important area. Indeed, the area of Emergences as studied by Hegel and Marx, would need to be applied within the regularly occurring alternation between long periods of Stability, the short interludes of Emergent Events comprising first a total dissociation of a prior Stability into something approaching almost complete chaos, then on into the remarkable Middle Phase, wherein both the Second Law of Thermodynamics and the very rarely observed opposite Law of Creative Constructive Order would have to be very deeply studied for many different areas, including most importantly Cosmology and the Creation and Evolution of Life on Earth, as well as Social Revolution, the First Appearance of Human Consciousness, and even Thought itself.

But, though clearly not personally adequately equipped for such a task, a start has to be made. The current persisting crisis in World Capitalism and the ever spreading tide of the Arab Spring, not to mention the increasing interventions in more and more countries by the leading Capitalist powers, make this worked increasingly urgent. History has demonstrated what happens in such circumstances.

Now, various areas have already been undertaken over the last decade, which will certainly provide the next initial steps in the right direction. I will, no doubt, be

severely criticised from the positions of many un-included specialisms, but I’m afraid vested interests are precisely what is NOT needed here. If you disagree, then you must make your contribution. Purely negative or current position-defending criticisms will be responded to mercilessly. The task is no longer defence, but to participate in the necessary Revolution.

Postscript: Diversionary Motives?

Though not directly concerned with Programming Languages, there has to be a final assessment as to just how the prevailing principles of the Society in which we live, have strongly diverted any truly revolutionary developments in I.T. Research. It, of course, could not be a general and comprehensive account, but even within the recent experiences of this researcher, there has been sufficient to demonstrate such effects very clearly.

The author’s long experience of work concerning Dance and Multimedia has already been mentioned elsewhere, but the gains made in that area, also had much wider implications. For to get anywhere a great deal of research had to be addressed concerning how movement should be accurately and appropriately captured via both video and film. And from this research, a whole host of entirely new techniques had to be invented to actually deliver all that was needed with accuracy and sensitivity, particularly in crucially expressive movements. And the contrast with stills being used inadequately to illustrate the articulations not only within given motifs, but also and vitally also between them in developing meaningful choreography.

The potential spin-offs in all movement-involved areas (such as most Sports) were evident to the team from the start (indeed, it was in that area that the original research was conceived of, but the most demanding area was certainly Dance, and there was a world class teacher, who was immediately interested in what was being proposed) So, both before and after the considerable work involving Dance, approaches were made and even demonstrations authored, to present to Cricket, Golf, Tennis, Diving, Gymnastics, and several other evident areas where the developed method could be extremely useful.

Indeed, the crucial turning point was in the publication of the 3-disc pack *Choreographic Outcomes* in 2005, which was brought to fruition in the following dance performance disc *Vocalise*. [All of these, plus another 10 titles, were published by Bedford Interactive Productions, following detailed researches by Bedford Interactive Research over the previous 15 years]

What was finally achieved in the ForMotion system was the simultaneous and synchronised appearance, on-screen, of different views of the same section of the Dance, under a single joint set of controls. A wide set of views to be switched between were available, so that particular interludes that required a different current pair of views could be easily switched to. In addition, more sophisticated features were made available such as to allow repeated looping around a movement or phrase. Perhaps the most powerful extra feature was that which allowed extractions to be made from still frames to enable animated overlays to be constructed, that could be applied back over the moving video, and synchronised to it. And both these were manipulatable jointly by all the usual controls.

Though the achievements in these two publications are now 6 years old, they are still far in advance of the overlays and techniques used in Cricket (as in the current Indian Premier League). The point of this brief postscript is that, in spite of these admirable and professional gains, NO Sport wanted to know, and the provision of presentations to experts in the I.T. field led to NO Company who wanted to implement such facilities as part of their multimedia Authoring Packages.

Profit was the evident primary motive with the companies, and jobs for ex-sport people were the motive in all the Sports we approached. It seems that bigger motivations than excellence dominate not only Society at large, but the many disciplines within it too.

S H A P E **JOURNAL** **E**

www.e-journal.org.uk