20 Years of Scripted Space

Malcolm McCullough is an advocate of the importance of play and manipulation within the parameters of established software. As he asserts, 'Once the design world has been set up, it still needs to be explored, played and mastered with finesse'. A veteran of architectural programming (he was the first architecture product manager for Autodesk, from 1985 to 1986), he reflects on the last two decades of development.

The Invitation: Rules and Two-Part Design

You have to get free of the grind. It is just too much work to construct every design element uniquely, directly and without regard for what knowledge it represents. Now that technology lets us treat abstract schemas as objects for manipulation, it makes no more sense to design by drawing each line and modelling every surface than it does to drive an aeroplane down a highway. The more kinds of representation that software lets us manipulate, the more opportunity we have to take design to a higher level. After all, the very essence of software is to represent problems abstractly, through the use of variables, conditionals, iterations and procedures. All of this has now been made accessible to nonspecialists via user-friendly, shrink-wrapped design software. The disciplined programming work has been done by the professionals behind all this gear: all you need is the will to improvise.

Indeed you must. For as the coders in Silicon Valley would be the first to admit, while their knowledge of shapes and data structures and usable interfaces naturally surpasses anything some casual tinkerer (or headstrong academic) could come up with on his or her own, such knowledge stops at the border between the theory and the application of form. They know how to process forms, but only the design professional knows which forms, when, where and why. Thus every discipline must bring its domain knowledge to the question of software representation. The software part has been made as easy as possible by new scripting languages. Architects, engineers, fabricators – any domain whose knowledge depends on form – have all begun to adapt and extend generic software tools to the specifics of their disciplines. It is no longer so rare for a design firm to have a few people writing code – not the kind of code that requires a degree in computer science to get right, but the kind that can be crafted one line at a time on top of commercial software while working on form. No need to learn how to link headers or throw exceptions. This kind of code is by you, for you – and it is fun.

This is because it lets you game the rules of play. The amazing part of scripting is how it adds a whole extra level to design thinking. First you set up some rules for generating forms, then you play them to see what kind of design world they create, and then you go back and tweak the rules. With a bit of interface technology, even just a few simple buttons sliders, you can tweak almost as quickly as you play.

This does require some change in outlook. Many designers, or at least most design students, believe that any constraints on the character and construction of form will just hamper their creativity. This is wrong as wrong can be. Any expressive medium has its idioms, types and genres, and the better established of those are often where the richest expressions occur. Meanwhile, many technologists and managers also believe that computer usage is serious work, and that play is just distraction or, worse, entertainment. This too is wrong. Even before computers, tangible speculation was the heart of design work. Software just articulates and accelerates that conjecture. We can ask 'What if?' more often, and about more abstract kinds of assumptions.

But consider one last misconception, namely the old ambition (especially prevalent in the 1980s) towards the computability of all knowledge. To understand the limits of this, it is a degree not in computer science but in philosophy that is necessary. Any sage should know that some things are easier done than said, that an approximation is almost always enough, that convenience of measurement should not be mistaken for completeness of truth, that logic is so cumbersome that humans seldom use it, and so on. Therefore the role of computers in design is seldom one of automation. There are few sets of design rules that can be set up to be

There are few sets of design rules that can be set up to be interesting enough to run on their own. Once the design world has been set up, it still needs to be explored, played and mastered with finesse.

interesting enough to run on their own. Once the design world has been set up, it still needs to be explored, played and mastered with finesse.

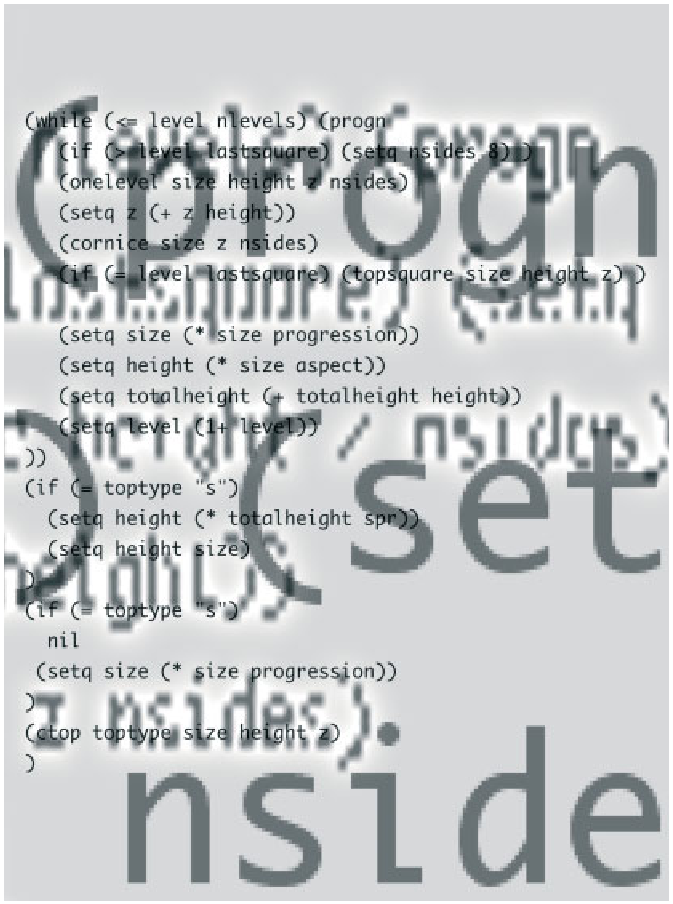
At least that is what they were saying in the 1980s. I'm old enough, and fortunate enough, to report some of this first-hand.

Some Basic History and Theory of Design Computing
Twenty years ago, in 1986, you couldn't just noodle with spline surface models all day. No online chat idled in some nearby window – nor shopping nor shooter game. No gargantuan operating system was there to interrupt you with security alerts, nor to push you unwanted software, nor to cram your 'personal' computer with 10,000 needless things. Back then there was no presumption that anyone but the computer was dumb. You could almost still hear yourself think.

That year is significant in terms of today's interest in scripting because it was the year Autodesk took over the personal computer design-software market – mainly, I still think, because of its new advantages in scripting language. When a desktop PC was the world's idea of a 'microcomputer', the field was still wide open: and while the big architecture firms were still breaking the bank on a Vax or a Prime, something had to give. The company's AutoCAD was the first of the PC computer-aided design (CAD) programs to record command-line sequences of its drawing operations (which remains essential to the craft aspect of this culture) within the variable structures and flow constructs of an interpreted programming language. And even if the vast majority of its users never fully rose to this, there was remarkable promise in that this language was Lisp, the usual preference among the artificial intelligence (AI) community of the day. If we define 'scripting' as coding one line at a time on an interpreter that belongs to a powerful host program, then it is difficult to find an earlier popular instance in design graphics, nor one that expanded the constituency for design computing so far, so rapidly. (Among my own responsibilities as the company's architecture product manager in 1986 was reviewing proposals from dozens of new businesses to be included in Autodesk's catalogue of such scripted 'third party' software.)

The universities were aiming much higher, of course. Also coming of age in the 1980s was a knowledge representation that has long remained an academic focus in architectural computing: shape grammars. (At Autodesk we read all the papers of the theory's inventor, George Stiny, with whom I had just studied briefly at UCLA.) Everyone knew that the discrete entity-and-layer structure of early CAD systems was not enough. Theoretically, grammars were a powerful way beyond this towards substitutions and subdivisions in articulating form. It was expected that these could become very practical expert assistants on well-formed classes of design problems. Unfortunately, the level of codification was high. Early applications operated within very specific architectural motifs, the most famous example of which was Wrightian Prairie-style houses. Even for applications in rote architectural production – say, laying out a set of hotel rooms – these better formal structures had less ready application to everyday architectural problem-solving.

The more accessible next step in knowledge representation was instead a dimensionally driven approach that became known simply as 'parametrics'. This is mainly a matter of expressing design problems or, more specifically, formal types, computationally in terms of a short set of independent design variables, especially dimensions. From the values of these few independent dimensions, software can derive a particular instance in that type of form. The essence of the type is implied in the parameterisation, the circumstances of the instance are specified in the arguments given to the dimensional variable, and the two-level process of design begins. Although some classes of form can be codified to a single size variable, such as is designated for a hat or a pair of shoes, most cannot. Although many more arbitrary classes of form can be constructed if the number of independent



```
(while (<= level nlevels) (progn
   (if (> level lastsquare) (setq nsides 8)) )
   (onelevel size height z nsides)
   (setq z (+ z height))
   (cornice size z nsides)
   (if (= level lastsquare) (topsquare size height z) )

   (setq size (* size progression))
   (setq height (* size aspect))
   (setq totalheight (+ totalheight height))
   (setq level (1+ level))
))
(if (= toptype "s")
  (setq height (* totalheight spr))
  (setq height size)
)
(if (= toptype "s")
  nil
 (setq size (* size progression))
)
(ctop toptype size height z)
)
```

Recalling AutoLisp, the earliest generative scripting language to be used widely in giving architectural form.

Tool-like operations modified visual objects of design in real time onscreen. Dense continuous notation became available for modelling, illustrating and orchestrating in time. Density was a matter of response time and resolution. Between any two states of a digital artefact, there was effectively another. This made it much easier to achieve design states by discovery through manipulation rather than derivation through formulas.

variables is sufficiently large, the parametric process is more valuable if that set of variables is kept manageably low. The value is in the use of dependent variables to generate relatively detailed instances from relatively few inputs. This is especially true if those inputs can couple the generation of form to the ranges and constraints of the machine processes by which they are to be fabricated.

'The essence of the architectural type' quickly became good subject matter in design education using the new computational medium. In 1989, a healthy few years in advance of the wave that would reduce us all to teaching freeform direct manipulation with the latest spline surface modeller, Harvard introduced what may have been the first programming course required for all professional degree candidates in a leading school of architecture. The software basis for this initiative was a program called TopDown, written mainly at UCLA, by Robin Liggett and William Mitchell. This had been inspired in part by shape grammars, and in part by the existing pedagogy of teaching design theory through programming, and in part by recent improvements in ready-made interface widgets. TopDown provided a visual and dynamic way to combine substitution and dimensional variations on a compositional motif. To make a beginning artefact in it involved in the order of 20 lines of Pascal code. Even this was alien, however.

A self-conscious and sleep-deprived Harvard architecture student was often not the best casual coder. Furthermore, much of the faculty then still believed that gentlemen did not operate machinery. It is an understatement to say that this course met with resentment.

The Direct Manipulation Boom

As the 1990s wore on, the graphical user interface did so much to make computing accessible to nonspecialists that it quickly became the only form of computer use that most people had ever known. Perhaps the most essential principle of this interface remained that 1980s breakthrough of 'direct manipulation'. Tool-like operations modified visual objects of design in real time onscreen. Dense continuous notation became available for modelling, illustrating and orchestrating in time. Density was a matter of response time and resolution. Between any two states of a digital artefact, there was effectively another. This made it much easier to achieve design states by discovery through manipulation rather than derivation through formulas. Nevertheless, a notation was effectively kept in software standard file formats. In other words, unlike any previous medium that was dense and fluid enough to allow continuous coaxing into conformity with whatever was held in the designer's mind's eye, this new medium had reproducible documentation. It also allowed a proliferation and management of versions without loss of quality. No longer 'just a tool' (in the apologetic sense of not influencing one's intentions), design computing rapidly matured as a medium in which bias, appreciation, expression and new genres were inevitable.

Alas the great rush to the seductions of all this new technical possibility effectively drowned most existing agendas in designers' programming culture. For the first time, the majority of computer users were noncoders. Improvisation prevailed over composition.

The theoretical prospects raised were of an open and bottom-up morphology rather than closed and top-down. Deleuzian difference-and-repetition spread through the discipline like an invasive species. So by 1996, not only did noodling with spline surfaces all day work comfortably in practice: it also worked in theory.

Of course there were other, more fundamental reasons why programming played out better in other form-giving disciplines than in architecture. The course of parametrics seems significant in this regard. Parametrics work better in domains whose subject matter is engineered form itself – especially in mechanical components for complex assemblies such as vehicles. Parametric design works less well where physical configuration and performance are just the means, and a more emergent usage pattern is the end. Or, to put it the other way round, when the subject matter of design is more the social arrangements and less the mechanical assemblies used to house them. Parameterisation breaks down when the design problems are wickedly under- or overconstrained, or where the design variables are less obvious. Compared to an aeroplane part, even the aforementioned rote hotel room is less computable.

Then, of course, the spirit of casual improvisation in code moved on – and struck gold with the World Wide Web. Accomplishments in scripting interactivity (for example using Java, Lingo and Perl) at least temporarily drew attention away from the prospect of scripting pure form. Indeed, it could make a million for you overnight.

Among those who did not abandon built form for more visual pursuits, code seemed unnecessary. Especially among architects preoccupied with radical novelty in autographic form – a pursuit now made so accessible by software that any fool could do it – the more immediate technical possibilities of direct manipulation were the order of the day. The most favoured genre of improvised objects became known as 'blobs'.

Nevertheless, the flame continued to burn for design education through programming. In 1996, John Maeda launched a Java-based design fundamentals pedagogy amid the gizmo-centrism of MIT, and the results were instantly stunning. As documented in *Design By Numbers*,[1] *Creative Code*[2] and the former's progeny online at dbn.media.met.edu, there is a lot to be said for the role of simple algorithmic beauty in aesthetic education. It is beyond the scope of this essay to document this landmark work, which is mentioned here mainly to set the historical context.

For one thing, the purity of Maeda's approach let aspiring designers step right outside the increasingly bloated commercial software standards. In the 1980s, the PC had been hailed as the first machine that someone might look forward to using, but by the late 1990s it was no longer so personal, nor pleasant, due to the disfiguring bloat of its operating system. Where any theory of abstract craft broke down was on the stability of the personal tools and medium. There wasn't any.

Even in the most delusional hours of the blobmeisters' boom, not all was the vagaries of fashion. And as the smoke cleared from the dotcom crash, many transformations in design work remained. Seemingly on the basis of such advances, programming culture has been rediscovered in architecture. Consider some reasons why.

Architecture Rediscovers Programming

Even in the most delusional hours of the blobmeisters' boom, not all was the vagaries of fashion. And as the smoke cleared from the dotcom crash, many transformations in design work remained. Seemingly on the basis of such advances, programming culture has been rediscovered in architecture. Consider some reasons why.

Advances in digital fabrication must be first among these motives. Simply put, there is now far more incentive to express design in terms of a few variables based on the machining process. Whether in the schools, design-build practices or new niches in the just-in-time supply chain for a rapidly building world, rapid prototyping and computer-numerically controlled (CNC) machining have become competitive necessities.

Second, the theoretical basis of cultural expression in form is increasingly informed by a domain of knowledge that appears relatively comfortable with notions of generative algorithmic beauty: namely biology. Especially with respect to growth and emergence, but also with respect to the harmonies and recursions of static biomorphic objects, this present fashion in architecture has a greater need for coded formulations.

In addition, more people (and most management gurus) know that information technology and organisational change are just two sides of the same coin. Task automation gives way to strategic reconfiguration that legitimises creative work in more circles, and creates niches for new kinds of practice in the delivery of customisable design.

And finally, and perhaps most widespread culturally, the crafts of personalising one's workspace and scripting one's intellectual pleasures have become far more distinct in the generation of designers who grew up with computing. The bloat does not seem to bother so many of them, and the interruptions and multitask, fragmented attention may actually be felt as an advantage. Some of these designers have excellent training in algorithmic structures, even.

Given the visual interfaces of better software today, with the right process mindset, you might not even know when you are coding. The trick is to see patterns, and then to find the free play within the structures of them. Surely this is a form of intelligence. AD

Notes

1 John Maeda, *Design By Numbers*, MIT Press (Cambridge, MA), 1999.

2 John Maeda, *Creative Code: Aesthetics and Computation*, Thames and Hudson (New York), 2004.