First Build Your Tools

Article · January 2013

DOI: 10.1002/9781118653074.ch2

1 author:

Robert Aish
University College London
52 PUBLICATIONS **560** CITATIONS

SEE PROFILE

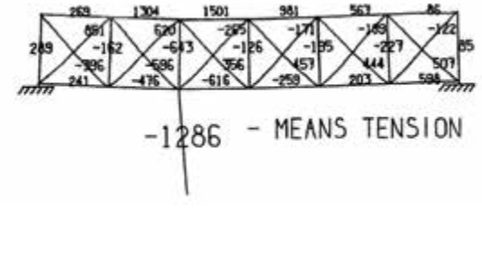
FIRST BUILD YOUR TOOLS

ROBERT AISH

One of the original members of Smartgeometry (SG), designer and software developer Robert Aish created software that responded to the vision of the SG community. In this chapter, Aish contextualises the history of SG within the development of computational design software. He explains that the contribution of SG has been to refocus the use of the computer towards design exploration rather than on production and downstream data management.

1

2



1 Ivan Sutherland's Sketchpad, 1963.
Real-time computation, graphical display, light pen interactions.

2 Ivan Sutherland's Sketchpad, 1963.
An example CAD model showing a truss under load.

The objective of the Smartgeometry (SG) initiative is to cultivate a thoughtful approach to design which combines advanced geometry and computation. Geometry and algorithms can exist in the abstract, but to be of any practical significance, to become a design tool which can be used by designers, then these have to be encapsulated in an executable form, as working software, hence the title of this chapter: 'First build your tools.' Tools do not exist in isolation. Tools require complementary skills to be effectively used. A computational tool requires cognitive skills. With this in mind we might summarise the mission of SG: to encourage the development of those cognitive and creative skills that matched the geometric and computational possibilities of a new generation of design software.

Much of the SG book will focus (correctly) on how designers are able to harness the ideas encapsulated in this software and express design concepts that otherwise would be difficult, if not impossible, to express. But occasionally (and this is one of those occasions) it may be interesting to go 'behind the facade', and understand some of the key ideas that contributed to the development of that software. We are going to focus on three ideas: Sketchpad, formal design methods and object-oriented software.

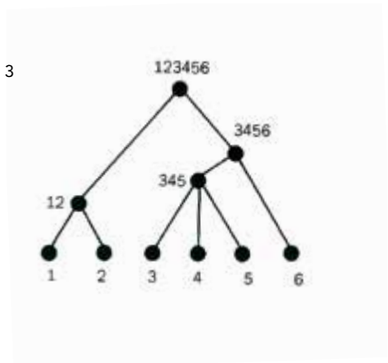
SKETCHPAD

The original computer-aided design (CAD) system was Sketchpad, developed by Ivan Sutherland in 1963.[1] The key features of Sketchpad are: an interactive real-time graphics display; an interactive input device (in this case a light pen); and an underlying model (in this case a constraint model). When the display and light pen are combined, they provide a complete interaction loop between the computer and the designer. The geometry displayed on the screen is a representation, not just of the apparent graphics (points, lines, etc), but also the underlying constraint model, which includes the symbolic representations of constraints (anchor points, parallel and orthogonal constraints between lines, etc). The manipulation of one geometric element by the user triggers a new resolution of the constraint model, which in turn propagates changes to the other geometric elements.

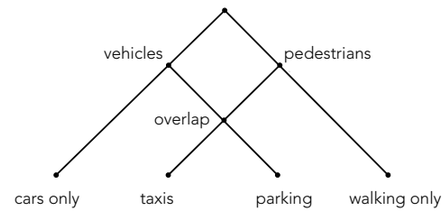
Sketchpad is a very early example of the model–view–controller (MVC) paradigm. The constraint system is the model, the display system is the view, and the light pen is the controller.[2] What is important is that the model, the view and the controller form a complete system and this is matched to the cognitive skills of the user, which taken together forms a man-machine system.
In 1967, I had the unique opportunity of using a direct successor to Sketchpad, the DEC PDP-7 at Imperial College, London.

This was a career-changing experience. The idea of a CAD as man-machine system, with an intelligent model linked by views and controllers to a designer, was fully established in the early 1960s. It would take a further 20 years to make this technology affordable and accessible to a wider audience.

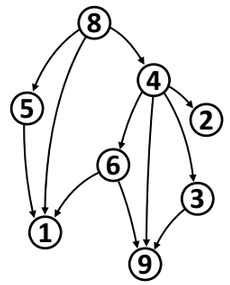
3



4



5

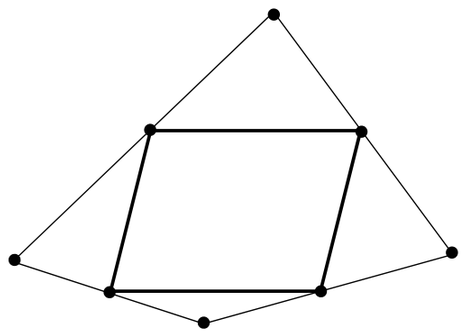


3 Christopher Alexander, tree diagram, from Alexander's 'A City is Not a Tree', first published and printed in two parts in the American journal Architectural Forum in April and May 1965.
The nodes of the graph represent components and assemblies and the arcs of the graph represent the 'contained in' relationship. The strict hierarchical decomposition of the tree structure only allows each subcomponent to be a member of a single higher-level (parent) assembly.

4 Christopher Alexander, illustration of a vehicular traffic system and a system of pedestrian circulation that overlap, from Alexander's 'A City is Not a Tree', first published and printed in two parts in the American journal *Architectural Forum* in April and May 1965.
The semi-lattice allows for overlaps, so that the same subcomponent can belong to multiple higher-level assemblies or subsystems.

5 A directed acyclic graph (DAG).

6 A constraint model (after Alan Borning).
For any given quadrilateral, joining the mid points of the side yields a parallelogram. Redrawn by the author, with permission, from Alan Borning, 'The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory', *ACM Transactions on Programming Languages and Systems*, Vol 3, No 4, October 1981, pp 353–87.



6

FORMAL DESIGN METHODS

Christopher Alexander is recognised as a pioneering design methodologist. Alexander's two main contributions to formal design methods are his book *Notes on the Synthesis of Form*[3] and his essay 'The City is not a Tree'[4]. Alexander was quite probably the first design theorist to propose that a hierarchical decomposition (as a formal *tree* structure) could be a useful way to understand a design problem or to describe a product. In his essay 'The City is not a Tree', Alexander rejected his initial use of the tree structure as being too restrictive because it failed to correctly represent the true complexity of relationships such as those he observed in the organisation of a city. He proposed the slightly more complex formalism of the *semi-lattice*.

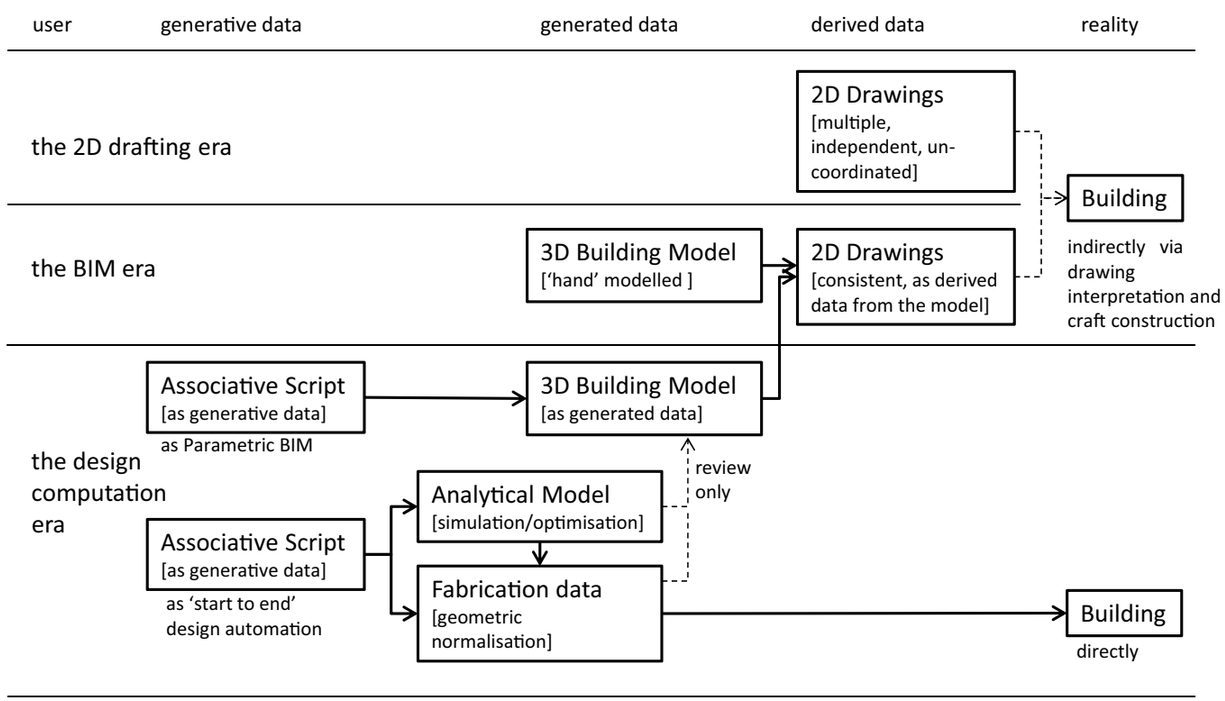
Alexander proposed using formal graph methods mainly as a descriptive technique to help designers understand the structure of resulting products or systems. Further research in the mid-1970s explored the use of the more general *directed acyclic graph* (DAG) to model design decision-making.[5] Here the nodes are the decisions and the arcs represent the directed relationships of a superior decision influencing or providing the context for a subsidiary decision.

The idea of representing a design problem as a direct graph is very general. It could be used to represent the *dependencies* between geometry, or between components, or even between more abstract ideas such as design decisions. The concept of graph-based dependency is quite understandable on a small scale. But for the designer (without any supporting software) it is difficult to apply on a scale that is appropriate to real-world problem solving. On the other hand, graph dependency is easily implemented as a program and can be applied to complex real-world tasks. Therefore graph dependency provides a common abstraction that can be shared between man and machine, with each playing a complementary role: one to define and the other to execute the graph. The idea of using a graph to present design dependencies was already established in the mid-1970s.

OBJECT-ORIENTED SOFTWARE

Object-oriented software emerged in the 1980s, pioneered by the development of the Smalltalk language at Xerox PARC in Palo Alto, California[6] and by researchers such as Alan Borning,[7] who used Smalltalk to develop ThingLab.

It is beyond the scope of this chapter to discuss the full range of concepts that are included in object-oriented (OO) programming (type system, inheritance, extensibility, encapsulation,



7 The three eras of CAD. The flow of information from the designer to the realisation of that data as a constructed building is tracked from left to right.

polymorphism, method overloading, etc). The key aspect of OO programming which is relevant here is the idea that the organisation of a program could more directly reflect the mental model of those involved. But this raises the questions: Whose mental model? That of the software developer or that of the program user? As I explained at the Object-Oriented Software Engineering Conference in London in 1990:

> *Object oriented software is based on a metaphor between real-world objects and computational objects. A specific assumption is made that a convenient ‘user model’ for software developers is one which mimics some of the attributes and behaviour of real-world objects. A CAD system is based on the inverse metaphor, that a convenient ‘user model’ for architects and designers is one where real-world objects (such as buildings) are represented as a computational system. An object-oriented CAD system is therefore a circular metaphor in which OO software concepts, derived from the real world, are re-applied to the design of further real world objects.*[8]

Therefore, when the OO software engineering principles are applied to the design of a CAD system, the inherent flexibility and extensibility that object orientation provides should be harnessed to benefit the designer. The use of OO software as the basis for computational design applications was already established in the early 1980s.

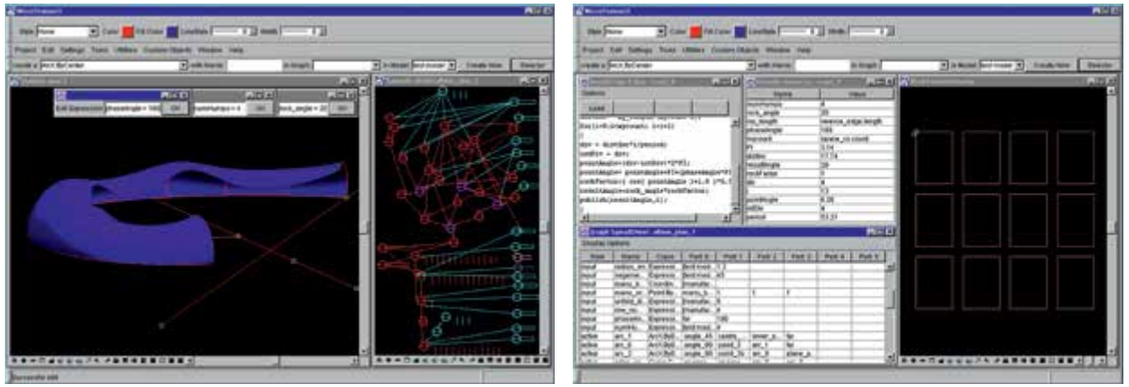
REVIEW OF SOFTWARE

We can summarise the history of practical CAD in terms of three eras: the 2D drafting era, the building information modelling (BIM) era and the design computation era in the diagram opposite These eras are recognisable but overlap in practice.

THE 2D DRAFTING ERA

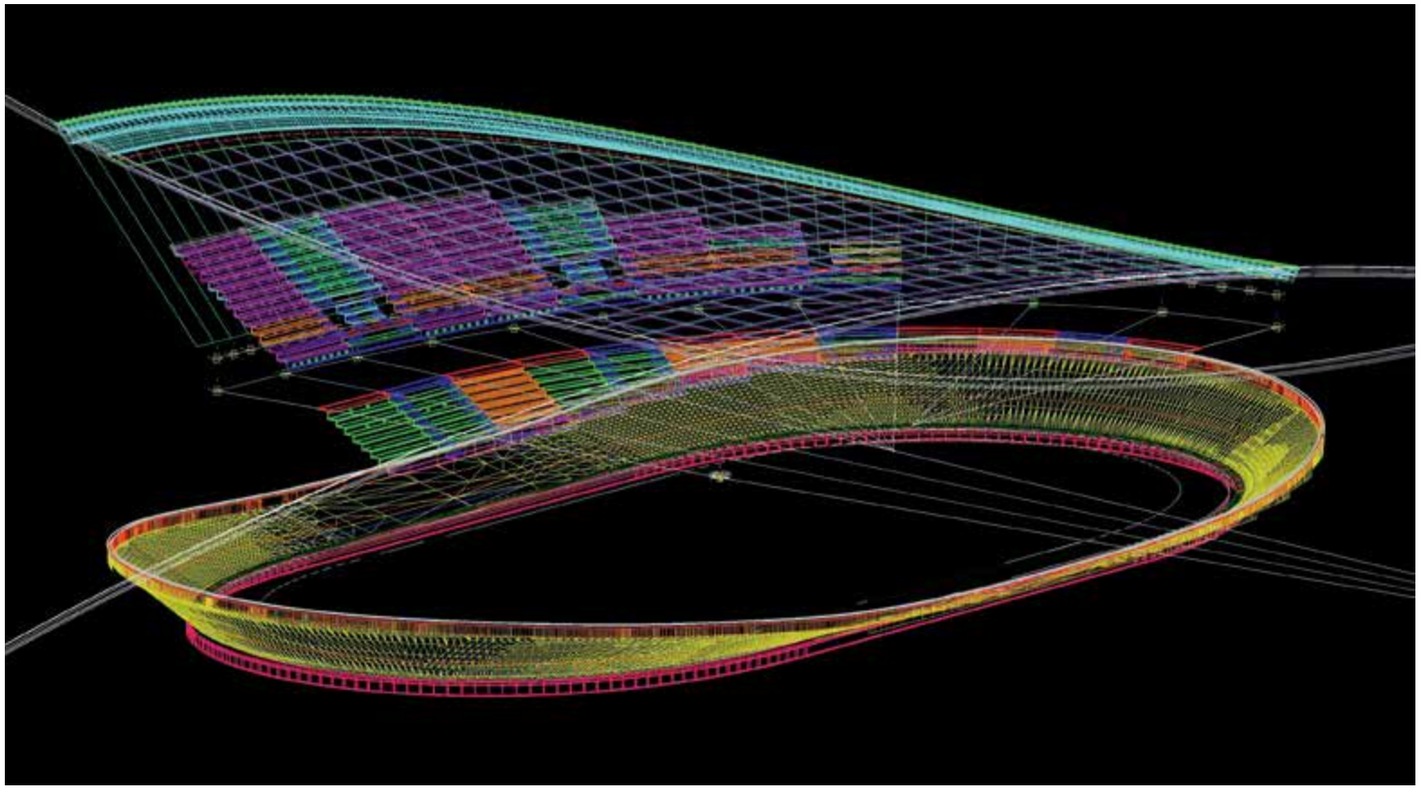
Starting in the early 1980s, 2D drafting continued the practice of representing buildings as multiple 2D drawings. 2D drafting technology could be retrofitted to existing design practice using existing skills without challenging established professional methods and conventions. 2D drafting looks very much like Sketchpad but with the constraint model left out. But the constraints were the whole purpose of Sketchpad. In retrospect, 2D drafting is really a travesty of Sutherland’s original intentions. The spectacular flaw of 2D drafting is that it failed to harness the potential of the computer as a creative design tool.

There are conflicting opinions about 2D drafting. Some have criticised 2D drafting systems as having a conservative influence on architecture because it perpetuated the use of drawings long after more advanced modelling technologies were available. Others have commented that the widespread adoption of 2D drafting could be claimed to be the ‘democratisation’ of technology. But the challenge is not how to ‘democratise’ the superficial aspects of a technology (in this case the graphic display and the input devices of Sketchpad), but



8

8 CustomObjects.
This was an important precursor to GenerativeComponents where the core ideas such as graph-based dependency and replication were originally developed. The model illustrated is the roof of Albion Wharf, designed by Foster + Partners, based on a parametric cosine curve. This CustomObjects model was a 'calibration' study to see how it could emulate the existing programmatic methods previously developed by Francis Aish and Hugh Whitehead at Foster + Partners.



9

9 Associative parametric model created using GenerativeComponents.
Extended caption?

how to 'democratise' the underlying concepts (in this case the constraint system): indeed the real challenge is how to 'democratise' thoughtfulness.

In fact it takes a real effort to re-establish the vision of the original innovators (such as Sutherland), that CAD is not a better way to draw but a deeper way to think. The contribution of SG can be viewed as part of a movement to put 'thoughtfulness' back into the use of computers in design.

THE BIM ERA

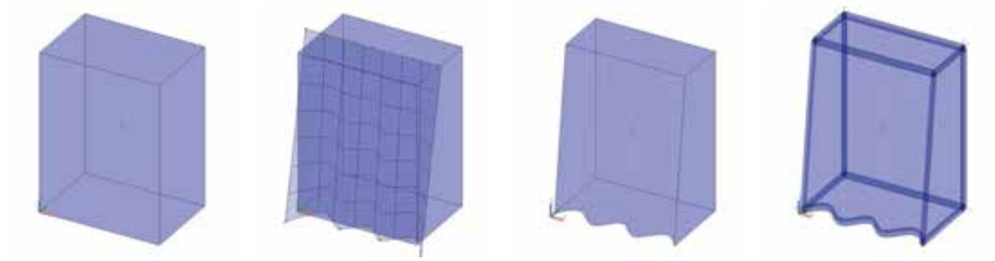
It may be surprising that the BIM era started before the 2D drafting era in the 1980s. One of the objectives of BIM is to overcome the limitations of 2D drafting that the description of the building is spread over multiple independent representations and there is no automated way to enforce consistency between these representations. BIM is based on the idea of creating a single 3D building model with drawings extracted from the model considered as derived data. BIM applies the principle of data normalisations to architectural modelling.[9] It is inherently a much more systematic approach to the application of computing to design. The BIM principles were already established in the mid-1980s: that is more than a quarter of a century ago.[10]

BIM assumes that buildings are assemblies of components, but that does not necessarily imply that a designer conceives of a building in terms of such assemblies. This 'component' assumption forces the designer to think about micro ideas (the components) before macro ideas (the building form). Some of the early 'proto' BIM systems – for example, MasterArchitect[11] – introduced a network of relationships between the components and used a special adaptive behaviour to make the editing of the architectural models more 'intuitive'. Again these rules presuppose not just a particular form of construction, but a particular way of design thinking. BIM has sometimes been criticised because it is extremely easy to create the obvious, but it is much harder to develop modelling innovations that do not follow the assumptions and hardcoded behaviour. BIM is a *technology*, combining data normalisation with geometric projection. BIM is a *methodology* which supports efficient integrated project delivery. But BIM should not be construed as a *philosophy* of design. It is not the reason for designing a building in a particular way.

The contribution of SG has been to refocus the use of the computer towards design exploration rather than on production and downstream data management.

THE DESIGN COMPUTATION ERA

Design computation introduced the distinction between a *generative* description of a building (as a graph or script) and the resulting *generated* model. The designer is no longer directly modelling the building: instead he develops a graph or script whose execution generates the model. This enables a completely different kind of architecture to be created. The design process also changes.



(a) Starting with a solid primitive

(b) Define a trimming surface

(c) Trim the solid with the surface to create the manifold building envelope

(d) Edges of the manifold building envelope as geometric supports for structural members

(e) Create horizontal, vertical and lateral ‘slicing’ planes

(f) Slice the manifold spatial building envelope to create a non-manifold spatial building model

(g) Edges of the non-manifold spatial building model as the geometric supports for curvilinear structural members

(h) Hide the non-manifold spatial building model to show the structural members

(i) Create a cone primitive to represent an atrium and use the ‘impose’ operation on the spatial building model

(j) Move the atrium so that it intersects with the spatial building model. New cells and edges are created

(k) As the atrium moves through the spatial model, new edges create related structural members

(l) Finally, the atrium is moved out of the spatial building model and adges and structural members adjust accordingly

10 The modelling sequence to construct a non-manifold spatial model of a building and the use of the edges of this model to construct a structural model.

An apparently minor edit to the graph or script could have a profound effect on the generated building, enabling the exploration of a vast array of alternatives. The objectives of design computation are to overcome many of the limitations of BIM: first, to move away from the manual model building and instead to directly harness program execution as a generative design tool; and second, to move away from hardcoded building semantics and allow the designer to create his own components and, more importantly, to define his own inter-component modelling behaviour.

A further motivation for developing the first generation of design computation applications (such as CustomObjects[12] and later GenerativeComponents (GC)[13]) was to combine the ideas from Sketchpad, formal design methods and object-oriented software concepts into a single system and to allow designers direct access to these ideas. During the first part of the design computation era, a number of related geometric concepts were also introduced: first, the use of the parameter space of curves and surfaces as a modelling context; second, providing the user with full access to transformation operations (shearing, scaling, rotation, translation); and third, the use of multiple model spaces, so that the same design could be represented in multiple configurations – for example, unfolded (for fabrication) and then folded (in situ).

The design computation era overlaps with the BIM era. Some design computation applications are used to generate BIM models (could we call this parametric BIM?).[14] The design computation era also saw the introduction of simulation and optimisation tools and the adoption of digital fabrication. It was apparent that analysis and simulation tools only required an idealised model of the building, and the data required to drive digital fabrication could be reduced, for example, to a tool-cutter path. The idea of ‘geometric normalisation’ was introduced.[15] Essentially, how could the designer build the lightest possible model, with the least effort, which would provide him with the most feedback, earliest in the design process? Why build a detailed BIM model (with all the extra effort and resistance to change) when an idealised, geometrically normalised model would be more appropriate?

The original tools of the design computation era (GC and Rhino Grasshopper) were directly based on a graph representation: indeed the graph was also the user interface. The visual graph approach is an excellent training device especially for novice users to construct simple associative models. Indeed, one of the major benefits of the graph as the user interface is that it has successfully attracted designers with little or no traditional interest or experience in programming. Only subsequently was the limitation of this approach appreciated when it was found that the visual graph approach does not scale to real-world complexity. We may convince ourselves that a picture is worth a thousand words, but after a while a graph diagram of hundreds of nodes becomes intractable. The architectural results may be geometric, but as we become more familiar with the programming concepts then a more compact description

11

11 The DesignScript IDE showing an associative graph model embedded within a user-defined imperative outer wrapper and iteratively driven, for example as an optimisation loop.

12

12 Hopkins Architects, London 2012 Velodrome, London, UK, 2012.
Extended caption?

as algebra (or program notation) eventually trumps node-based diagramming. Although the associative graph approach is well suited to the capture (and re-execution) of some geometric modelling operations, it is outside the mainstream of computing, which is dominated by imperative scripting and programming languages. Indeed many of the familiar concepts of imperative programming, such as iteration, cannot be expressed with a directed graph.

The objective of the second generation of design computation tools (such as DesignScript) is to overcome many of these limitations by completely integrating associative and imperative scripting and by reducing the reliance on the visual graph as the principle 'driver' interface.[16] These second-generation tools have also introduced topological and idealised representations[17] that are more suited to the geometric normalisations required for simulation and optimisation.[18]

Programming (as with exploratory design) is all about control. Different styles of programming use different approaches to *flow control*. Imperative programming is characterised by explicit 'flow control' using *for* loops (for iteration) and *if* statements (for conditionals) as found in familiar scripting and programming languages such as Processing or Python. On the other hand, associative programming uses the graph dependencies to establish 'flow control'. Changes to 'upstream' variables automatically propagate changes to downstream variables, as found in data flow systems such as GC, Rhino Grasshopper and Max/MSP. Imperative programming is appropriate for conventional scripting, while associative programming is an appropriate way to represent complex geometric modelling operations. Any serious design project is likely to require both approaches. Therefore it is highly desirable that the border between imperative and associative programming be as porous as possible. In DesignScript this is achieved by using a common programming notation for both imperative and associative programming.[16]

CONCLUSIONS

In retrospect the history of SG appears to be a perfectly logical progression. New innovations triggered the development of successive waves of software which addressed the limitations and conceptual mismatches evident in earlier software. But there was nothing inevitable here. There were false starts. Tangents were followed. Blind alleys were explored. Software platforms were developed and cancelled. Risks were taken. Applications were written and rewritten and rewritten. The midnight oil was burnt.

The key contribution of SG was to create an environment where designers (as software users) and software developers could share ideas. The emphasis was on exploration outside of the conventional, on abstractions beyond the familiar, on deliberately embracing cognitive retooling, on intentionally searching for integration across discipline boundaries, and on focusing on the ultimate quality and thoughtfulness of design rather than

on immediate productivity. Whatever the initial hesitation and however tortuous the path, the final results speak for themselves.

This leads to two important conclusions. First, computational design is now recognised as an intrinsic aspect of design creativity so that the mark of the accomplished designer is that he can move effortlessly between intuition and logic. Second, the distinction between the role of the professional software developer and the designer (as an end-user programmer) has been blurred. Now we can all create that final layer of scripting that represents our unique design logic, so that we can all say: 'Before I design I will first build my tools.'

REFERENCES

1 Ivan Edward Sutherland, 'Sketchpad: A Man–Machine Graphical Communication System', *AFIPS Conference Proceedings*, Vol 23, 1963, pp 232–8.
2 Trygve Reenskaug, 'Models-Views-Controllers', http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html [accessed 5 October 2012].
3 Christopher Alexander, *Notes on the Synthesis of Form*, Harvard University Press (Cambridge, MA), 1964; Christopher Alexander, 'A City is not a Tree', *Design*, No 206, Council of Industrial Design, (London), 1966 (online at http://www.rudi.net/pages/8755 [accessed 5 October 2012]).
4 Robert Aish, *An Analytical Approach to the Design of a Man–Machine Interface*, PhD thesis, University of Essex,1974.
5 Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley (Reading, MA; London), 1982.
6 Alan Borning, 'The Programming Language Aspects of ThingLab: A Constraint-Oriented Simulation Laboratory', *ACM TOPLAS*, Vol 3, No 4, October 1981, pp 353–87.
7 Alan Borning, 'ThingLab Demonstration Video', presented at The Computing Form, Xerox PARC, Palo Alto, CA, 1978.
8 Robert Aish, 'A User Extensible Object-Oriented CAD System', *Proceedings of the Object-Oriented Software Engineering Conference*, British Computer Society (London), 1990, pp **–**.
9 Edgar Frank Codd, 'Is Your DBMS Really Relational?', *Computerworld*, Oct 14, 1985.
10 Robert Aish, 'Building Modelling: The Key to Integrated Construction CAD', *Proceedings of the Fifth International Symposium on the use of Computers for Environmental Engineering related to Buildings*, CIB, 1986, pp **–**.
11 Robert Aish, 'MasterArchitect: An Object-Based Architectural Design and Production System, *Proceedings of the CIB W74 + W78 Seminar*, Lund, Sweden, 1988, pp **–**.
12 Robert Aish, 'Extensible Computational Design Tools for Exploratory Architecture', in Branko Kolarevic (ed), *Architecture in the Digital Age: Design and Manufacturing*, Taylor & Francis (location), 2003.
13 Robert Aish and Robert Woodbury, 'Multi-Level Interaction in Parametric Design', in Andreas Butz, Brian D Fisher, Antonio Krüger and Patrick Olivier (eds), *Smart Graphics, 5th International Symposium*, Springer (location), 2005, pp **–**.
14 Martha Tsigkari, Adam Davis and Francis Aish, 'A Sense of Purpose: Mathematics and Performance in Environmental Design', in *Mathematics of Space – AD* (*Architectural Design*), Vol 81, No 4, 2011, pp 54–7.
15 Fabian Scheurer and Hanno Stehling, 'Lost in Parameter Space?', in *Mathematics of Space – AD* (*Architectural Design*), Vol 81, No 4, 2011, pp 70–79.
16 Robert Aish, 'DesignScript: Origins, Explanation, Illustration', *Design Modelling Symposium*, Springer (Berlin), 2011, pp **–**.
17 Robert Aish and Aparajit Pratap, 'Spatial Information Modelling of Buildings using Non-Manifold Topology with ASM and DesignScript', *Advances in Architectural Geometry*, Paris, 2012, pp **–**.
18 Robert Aish, Al Fisher, Sam Joyce and Andrew Marsh, 'Progress Towards Multi-Criteria Design Optimisation using DesignScript with SMART Form, Robot Structural Analysis and Ecotect Building Performance Analysis', *ACADIA*, 2012, pp **–**.