DesignScript: Scalable Tools for Design Computation

Robert Aish
Autodesk, UK
http://labs.autodesk.com/utilities/designscript/, http://www.designscript.org/
Robert.Aish@Autodesk.com

Abstract. *Design computation based on data flow graph diagramming is a well-established technique. The intention of DesignScript is to recognise this type of data flow modeling as a form of 'associative' programming and to combine this with the more conventional 'imperative' form of programming into a single unified computational design application. The use of this application is intended to range from very simple graph based exploratory 'proto-programming' as used by novice end-user programmers to multi-disciplinary design optimisation as used by more experienced computational designers.*
Keywords. *Graph; scripting; associative; imperative.*

INTRODUCTION

The development of DesignScript is intended to be a response to the following trends in contemporary design practice (Figure 1). These trends can be described in terms of the following dimensions.

Scalable to different Computational skills

It is generally recognised that there are advantages in making design computation more accessible to a wider audience of designers. As software developers, we can interpret this not just as the need to make computational design tools easier to use by non-experts. In addition, there is an assumption or indeed an expectation that once designers have successfully accomplished some initial tasks, they will be interested in progressing from an exploratory approach to more formal programming methods. Therefore, we need to step back from the immediate requirement (to make computational design tools more accessible) and consider a more general requirement which is to support a progression in the development of

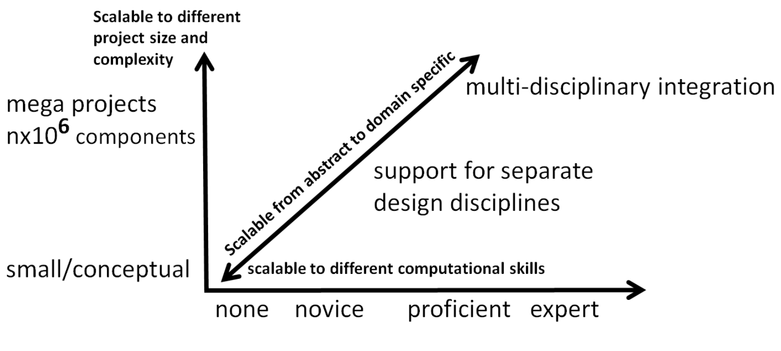

Scalable to different project size and complexity
mega projects
nx10^6 components
small/conceptual
Scalable from abstract to domain specific
multi-disciplinary integration
support for separate design disciplines
scalable to different computational skills
none
novice
proficient
expert

Figure 1
The three dimensions describing critical aspects of contemporary design practice.

computational skills, from 'no skills' to novice skills and hence to more proficient and expert skills.

This idea of a 'progression in computational skills' is based on the generally recognised critique of graph diagramming methods. While graph diagramming is an extremely powerful technique to enable novice programmers to create their first computational models with the minimum of experience and skill, it is generally recognised that the graph node representation does not scale to more complex logic. Indeed the visual complexity of the graph may become overwhelming and counter productive. It is exactly at this point where the application should encourage the novice programmer to make the transition from graph node diagramming to scripting: literally from 'node to code' (Figures 2-4).

Initially such scripts will reflect the graph node style of exploratory programing but as the designer continues to enhance and extend these scripts, the applications should help him to progress to a more formal style of software engineering by providing such facilities as a 'type' system and tools to support the refactoring of scripts into functions and classes.

An interesting by-product of a design application that supports different levels of skills is that it can encourage a more effective collaboration between team members with different skill levels.

Scalable from abstract to domain specific, including the support for multi-disciplinary design integration

While much of existing design practice is based on different disciplines (architecture, structural engineering and environmental performance, etc.) there are generally recognised advantages if a more holistic approach is adopted that integrates different ways of design thinking. This approach to design is often exercised through the formation of multidisciplinary design teams and can be encouraged by software that combines multiple analytical and simulation methods (Figure 5).

Another characteristic of innovative architecture practice is a 'return to first principles' often demonstrated through architectural form or engineering innovation. Often these principles may not have been previously associated with architecture or supported by existing design applications. This suggests a two way progression:

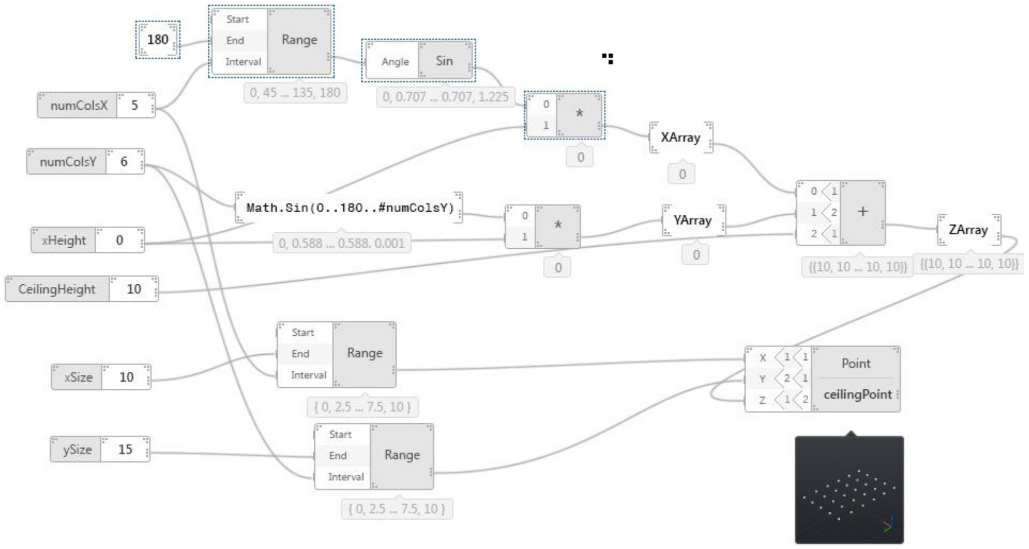


Figure 2
Starting with a conventional graph node diagram…

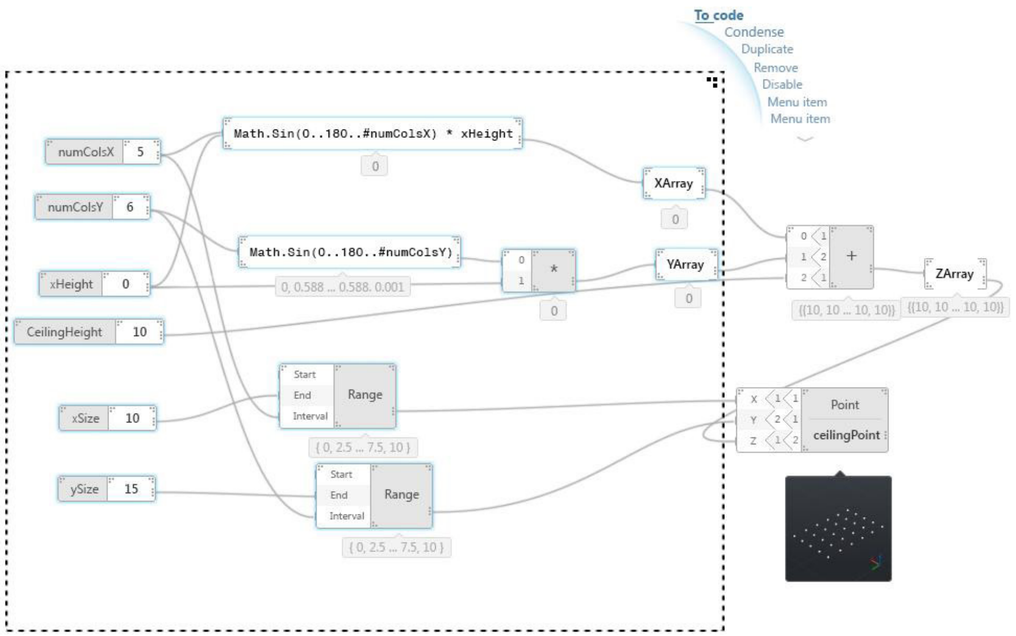
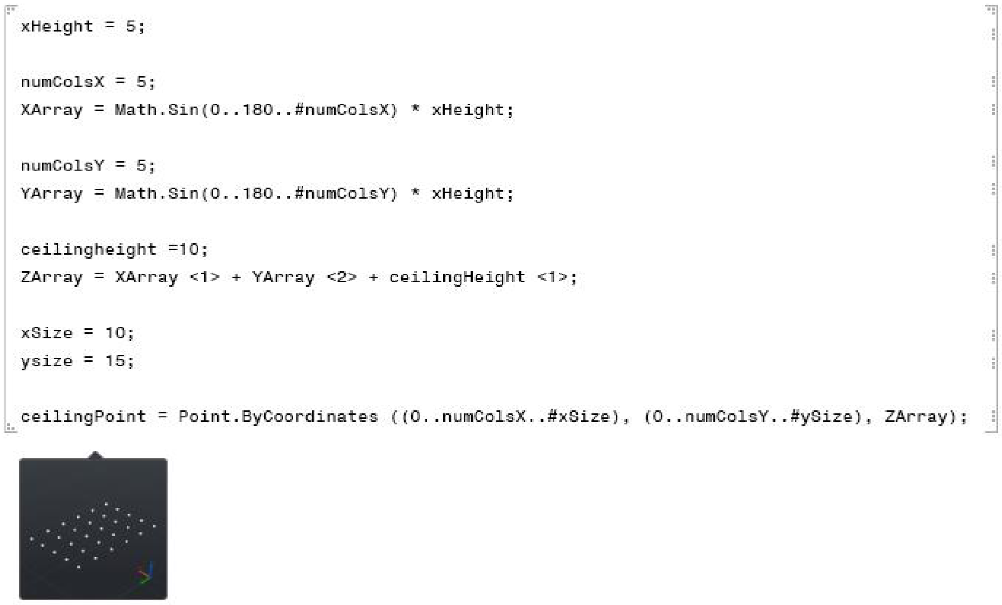


Figure 3
The designer select a region of the graph and use the marking menu…



```
xHeight = 5;

numColsX = 5;
XArray = Math.Sin(0..180..#numColsX) * xHeight;

numColsY = 5;
YArray = Math.Sin(0..180..#numColsY) * xHeight;

ceilingheight =10;
ZArray = XArray <1> + YArray <2> + ceilingHeight <1>;

xSize = 10;
ysize = 15;

ceilingPoint = Point.ByCoordinates ((0..numColsX..#xSize), (0..numColsY..#ySize), ZArray);
```

Figure 4
To turn the "nodes into code"…

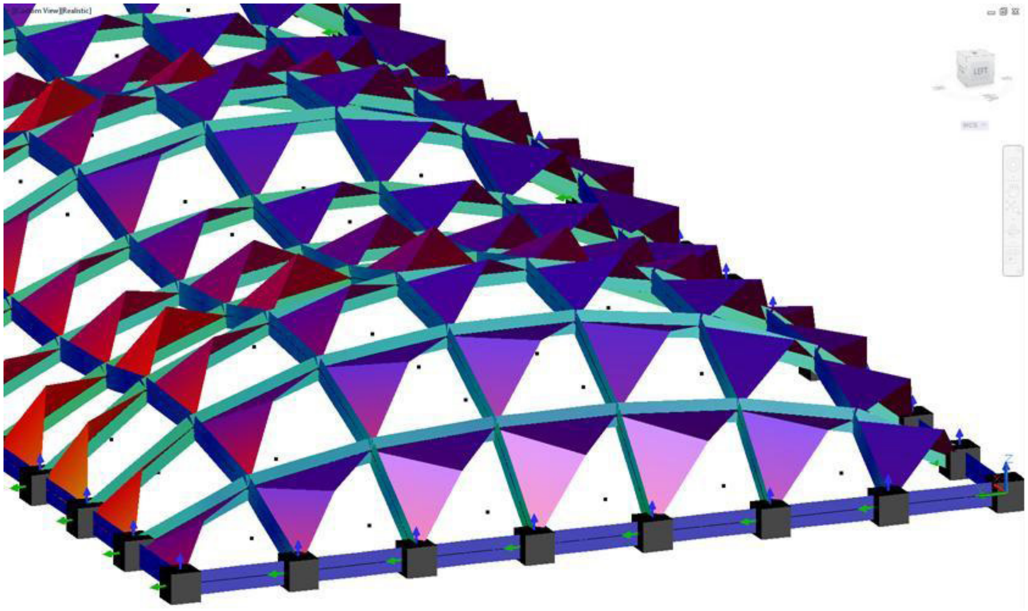


Figure 5
An example of a multi-disciplinary design model created with DesignScript, using a mix of simulation and analytical techniques provided by the different available plug-in's, including: form finding [with SmartForm], structural analysis [with Robot] and insolation on the shading devices [using Performative Design]. The colour coding on the structural members indicates structural utilization, while the colour coding on the shading devices represents insolation. Note: the structural nodes on the periphery are fixed, the ones in the centre are free to find their correct position.

On the one hand there are advantages in having design applications support more domain specific functionality: not in isolation, but rather integrated into a single common application framework and capable of acting as a computational intermediary between the different members of a multi-disciplinary design team.

On the other hand there are advantages if design applications could step back from being too domain specific and support a 'return to first principles' by exposing both computational and geometric abstractions directly to the designer.

Scalable to projects of different size and complexity

Design concepts often start as disarmingly elegant and simple ideas which can easily be explored with lightweight models or scripts. But to be realised as a physical building, these ideas necessarily have to be developed into complex building models with hundreds of thousands of individually detailed components.

In some cases the 'concept' is the overall form of the building, but increasingly the design concept may be an engineering principle or an objective to achieve a particular combination of performance metrics, or the use of a particular generative algorithm distributed within the individual components. In these latter approaches to the design, the architectural 'form' may emerge as a consequence of the design process rather than imposed 'top-down'. In addition, the resulting design representation may not be a traditional 'building model' primarily intended to support conventional drawing extraction. Rather it may be a series of 'geometric normalisations' intended to be the minimum information required for a direct digital fabrication process or robotic construction. Indeed conventional workflows (or data flows) are being supplemented by innovative project specific processes. Because of the essential 'open-endedness' of this new form of design, it is increasingly important that design applications are similarly 'open-ended'.

Summary

In summary, DesignScript is a computational system which is intended: to support the progressive acquisition of computational skills, to encourage the integration of different design disciplines and to support projects of different size and complexity.

Whatever changes occur it is important that the computational application can accommodate these changes and maintain a common underlying representation of design logic with no loss of fidelity, capability or performance.

DESIGNSCRIPT AS A DOMAIN-SPECIFIC LANGUAGE

Graph node diagraming is now a well established technique in design computation. The intention of a graph node diagramming user interface is to provide the designer with an intuitive and easily used 'proto-programming' tool that requires little or no previous understanding of programming concepts. Many of the existing applications which support a graph node style of computational design have made a separation between this type of interactive dependency modeling and regular scripting using conventional imperative languages such as Python, C# and Java.

Effectively, graph node diagramming is a form of 'data flow' programming which has developed independently of conventional programming languages. The intention of DesignScript is to explicitly recognise graph-based dependency modelling as a form of 'associative' programming. We can define an associative programming language as one which represents data flow in a human readable text notation. DesignScript supports the display and interaction with the underlying dependencies via both graph node diagramming and a corresponding text based associative language.

Indeed, while the designer is using graph node diagramming he may not be aware of that the graph is being recorded in DesignScript. But as the designer progresses and wishes to explore more complex modeling tasks he will necessarily need to acquire a more formal understanding of programming and design computation. The 'node to code' transition process is an extremely powerful way of using the result of an initial 'graph modeling' as the starting point for scripting, but there are still some challenges facing the designer to acquire the underlying programing concepts if he is to harness more of the potential of scripting. The following discussion might serve as an introduction to these issues and extends the previous work (Aish, 2011; 2013).

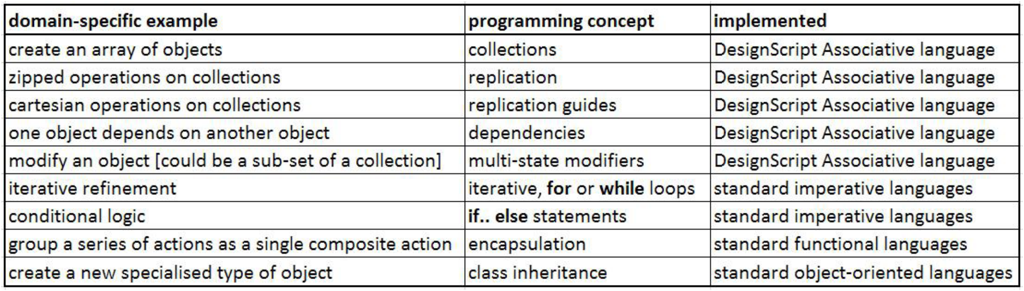
Within a domain specific application, such as DesignScript, we need to distinguish between the functionality related to particular 'domain-specific' objects such as walls, windows, columns and beams which typically would be implemented in specialised application libraries, and more general functionality that would be implemented at the language level. In this context a domain-specific language implements many of the facilities found in general purpose programming languages. In addition, a domaim specific language takes more general concepts from the application domain and promotes these concepts to be 'first class' features of the language (Table 1).

DesignScript could be described as a hybrid language which implements familiar concepts (and syntax) found in imperative, functional and object oriented languages and combines these with a number of new innovations in the form of an associative language.

The essential hybrid 'associative-imperative' nature of DesignScript is illustrated in the way it combines two domain specific ideas. First, the idea of a building being composed of a series of dependent collections of components where some members have special conditions' (typically supported by graph node diagramming and its representation as an associative language). Second, the idea of 'designing' as an exploratory activity which involves iterative refinement (as supported by iteration and conditional logic found in conventional imperative languages).

The differences between Associative and Imperative programming

Both Associative and Imperative languages have executable statements, for example:



domain-specific example	programming concept	implemented
create an array of objects	collections	DesignScript Associative language
zipped operations on collections	replication	DesignScript Associative language
cartesian operations on collections	replication guides	DesignScript Associative language
one object depends on another object	dependencies	DesignScript Associative language
modify an object [could be a sub-set of a collection]	multi-state modifiers	DesignScript Associative language
iterative refinement	iterative, **for** or **while** loops	standard imperative languages
conditional logic	**if.. else** statements	standard imperative languages
group a series of actions as a single composite action	encapsulation	standard functional languages
create a new specialised type of object	class inheritance	standard object-oriented languages

Table 1
DesignScript as a 'domain-specific' languages implements many of the facilities found in general purpose programming languages and adapts these for a domain-specific purpose and in addition takes more general concepts from the application domain and promotes these concepts to be 'first class' features of the language.

a = 10;
*b = a * 2;*
.
.
a = 20;

In the case of an imperative language, the final statement *a = 20;* does not cause the previous statement *b = a * 2;* to be re-executed.

In the case of an associative language, the statement *b = a * 2;* is not just an executable statement, it is also records a persistent dependency relationship between variables *b* and *a*, such that any change to *a* will automatically force a re-execution of all statements where *a* is referenced on the right hand side, as in *b = a * 2;*.

Imperative programming uses special 'flow control' statements, such as **for** and **while** loops and **if .. else** statements. These flow control statements are independent of the executable statements. In the absence of any flow control statements imperative programming executes statements in the sequence of the source code (i.e. in the lexical order). Associative programming does not have separate flow control statements but instead uses the concept of dependencies inherent within each statement (such as to *b = a * 2;*) to create a topological ordering of all the statements. The statements are then executed in this topological order.

In addition, associative programming within DesignScript also introduces two further concepts: 'replication' (in various forms) and 'modifiers'. The full power of DesignScript emerges when the graph based dependencies, replication and modifiers are all combined. These are discussed in the following sections.

Replication

In many existing programming languages a distinction is made between a single variable of a particular type and an array or collection of that same underlying type. This distinction restricts the interchangeable use of a collection or a single value of the same underlying type and forces the programmer to write different code for a single variable and for arrays or collections.

One of the domain-specific aspects of DesignScript is to relax this restriction and make the language more flexible and more tuned to its use by novice programmers. In this context, DesignScript introduces the concept of 'replication'. With replication, anywhere a single value is expected a collection may be used instead and the execution of the statement containing the collection is automatically executed for each member in that collection.

The combined result of dependencies and replication is that it is easy to program complex data flows (including geometric operations) involving collections. An upstream variable may change from being a single value to a collection or from a collection to another collection of different dimensions or size. As a consequence, the downstream dependent variables will automatically follow suit and also become a collection of the appropriate dimension and size. This makes associative programming incredibly powerful, particularly in the context of generating and controlling design geometry. It avoids the designer (as a novice programmer) from being pre-occupied with the size or dimensionality of variables.

Zipped replication
When there are multiple collections within the same expression we need to control how these are combined. With 'zipped' replication, when there are multiple collections, the corresponding member of each input collection is used for each evaluation of the expression. This works well when all collections are the same dimension and length. If collections are of different lengths, then the shortest collections determines the number of times the expression is evaluated, and hence the size of the resulting collection.

```
a; b; c;    // define the variables to be output at the top
or outer scope
 [Associative]
{
   a = {1, 5, 9};
   b = {2, 4, 6};
   c = a + b;    // zipped replication operation .. c = {3,
9, 15}
}
```

The equivalent imperative code would be:

```
a = {1, 5, 9};  // define the variables to be output at the
top or outer scope
b = {2, 4, 6};
c;
[Imperative]
{
   n = Math.Min(Count(a), Count(b));
   for (i in 0..n)
   {
      c[i] = a[i] + b[i];
   }
}
// c = {3, 9, 15}
```

Cartesian replication
When there are multiple input collections we need to control how these are combined. With 'cartesian' replication, all members of all collections are evaluated so that resulting collection is the 'cartesian product' of the input collections.

DesignScript introduces a special notation called 'replication guides' to control the order in which the cartesian product is created and takes the form <*n*>, where *n* defines the sequence of the replication operations. This sequence is equivalent to the order of the nested *for* loops that would have had to be written in an imperative script.

```
a; b; c; d; // define the variables at the top or outer sco-
pe
[Associative]
{
   a = {1, 5, 9};
   b = {2, 4 };
    c = a<1> + b<2>;  // cartesian replication  c = { { 3, 5
}, { 7, 9 }, { 11, 13 } }
   // changing the sequence of replication guides chan-
ges the resulting collection
   d = a<2> + b<1>;  // d = { { 3, 7, 11 }, { 5, 9, 13 } }
}
```

The equivalent Imperative code would require doubly nested *for* loops. To compute *c* the outer loop would iterate over *a* and the inner loop would iterate over *b*.

```
a = {1, 5, 9}; b = {2, 4, 6}; c; d;  // define the variables at
the top or outer scope
[Imperative]
{
   m = Count(a);   n  = Count(b);
   for (i in 0..m)
   {
      for (j in 0..n)
      {
         c[i][j] = a[i] + b[j];
         d[j][i] = a[i] + b[j];
      }
}
// c = { { 3, 5 }, { 7, 9 }, { 11, 13 } }
// d = { { 3, 7, 11 }, { 5, 9, 13 } }
```

Modifiers
With modifiers each variable can have multiple states which might be used to create a geometric modelling sequence. For example a geometric

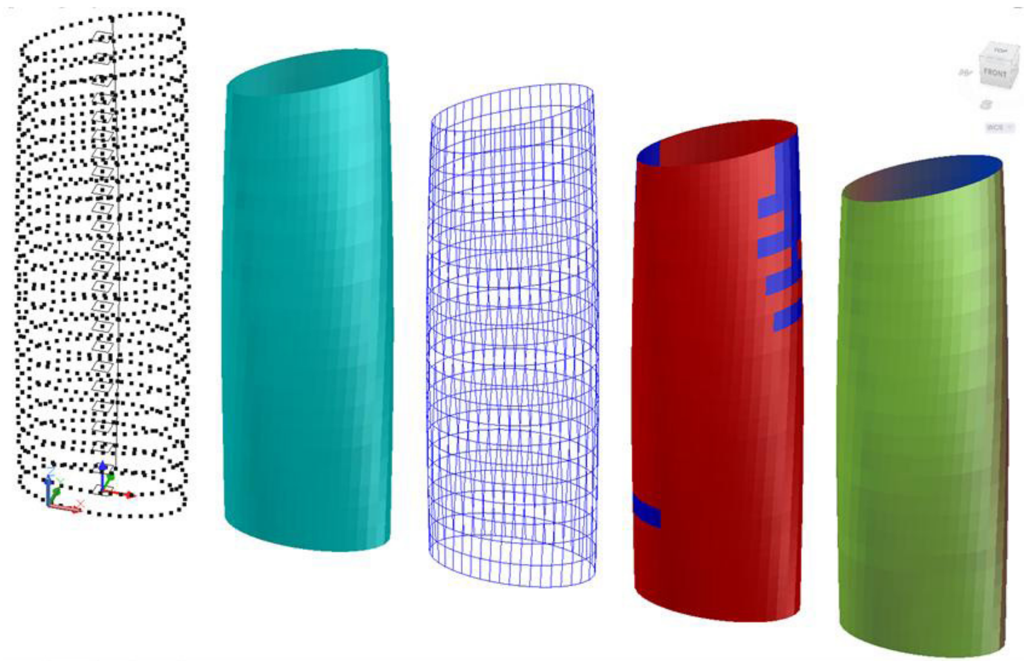


Figure 6
Façade design and analysis with DesignScript, showing the development of a conceptual building model as a series of polygons. Those polygon which are significantly 'out of plane' can be visually identified [indicated in blue in the model, second from the right] and this condition can be used in a downstream fabrication process, for example by switching to different types of facade panels.

variable might be created (say as a curve) and then 'modified' by being trimmed, projected, extended, transformed or translated. Without the concept of modifiers each state or modelling operation would require to be a separate variable and this would force the user to have to make up the names of all these intermediate variables. Modifiers avoid imposing this naming process on the user.

Combining dependencies, replication and modifiers

Dependencies, replication and modifiers can all be combined to represent the typical modeling operations found in architecture and constructions. Buildings are composed of collections of components. Typically these collections are often the product of a series of standard operations across all members. On the other hand, within such collections there may be special conditions where different or additional modeling operations are required to be applied to a sub collection of members. Modifiers enable these special conditions to be identified and the additional modelling operation applied.

To give an example, imagine that a building façade is based on a set of polygons. The polygons will be the 'support' geometry for the façade panels. However, in this example those polygons which are 'out of plane' by some critical dimension require a special modification before being used as the support for the corresponding facade panels (Figure 6).

The designer may want to apply a special operation but only to the 'out of plane' polygons and the application of this operation should not alter the particular polygon's membership of the collection of polygons. In this context all the polygons have a common defining operation, but some polygons have an additional 'modifier' operation applied.

Having applied the special modifier operation to the specific polygons, the designer can use the whole collection of polygons to generate the collection of façade panels. The collection of façade panels is dependent on the heterogeneous collection of polygons. Additionally, the designer may want to embed the whole polygon collection and façade panel generation process in an iterative loop to optimise the design for some criteria, such as solar gain, structural efficiency.

While this example is reasonable 'domain-specific' (dealing with polygons and facade panels), the fundamental ideas (of collections, modifiers and dependencies) on which the DesignScript language is based are necessarily quite abstract.

CONCLUSION

DesignScript is intended to address many of the critical issues in contemporary design practice, including the progressive acquisition of computational skills, the shift to multi-disciplinary integration and scalability of projects.

At one level DesignScript is an intuitive application which can be used by designers (as novice programmers) with the minimum programming prerequisites. Yet behind this intuitive user interface is a highly innovative programming language that introduces a number of domain-specific programming ideas including associativity, replication and modifiers. These innovative ideas are combined with well established conventions drawn from imperative, functional and object oriented programming languages and unified into a single scalable computational design application. A special characteristic of the user interface is that it supports the progression from novice user to more accomplished programmer by progressively revealing or unmasking these underlying computational concepts.

REFERENCES

Aish, R 2011, 'DesignScript: origins, explanation, illustration', *Proceedings of the Design Modelling Symposium*, Springer, Berlin.

Aish, R 2013, 'DesignScript: a Learning enviroment for Design Computation', *Proceedings of the Design Modelling Symposium*, Springer, Berlin.