Chapter 4

Programming

Algorithms are realized as *programs*, which in turn are written in precise and prescribed *programming languages*. Almost universally, designers learn to think algorithmically by learning a programming language to accomplish design work. Anyone who has become a good programmer will tell you that, at some time, they focused intensely on programming and spent a great deal of time learning to do it well. The term "programming language" itself gives a hint as to why this is so. Just as the most effective way to learn a new natural language is to immerse yourself in the daily life of native speakers of that language, the best way to learn programming is to work intensely with a programming language to the near exclusion of other forms of thought. But even when a designer has expended all of this effort and has become an accomplished programmer, there will be aspects of algorithmic thinking still unlearned. This happens because there are general language-independent concepts to master. Computer scientists learn these through a combination more abstract algorithmic descriptions and programming in multiple languages. In fact, most computer scientists would assert that the science of computing is quite separate from programming skill. In particular, they will claim that not all programmers are computer scientists and (more reluctantly) that not all computer scientists are programmers. Like drawing in design, programming is the craft skill by which much of computer science is done.

Programming is not a monolith. It is a multi-faceted skill mostly taught in sequence. At every stage in the sequence you rely on concepts learned earlier. Importantly, at every stage you can accomplish *some* work with what you know. Almost all books and courses on programming languages progress through such a sequence of concepts each realized as short programs. Sequences are startlingly similar across books, courses and languages. Every programming language is, in fact, a concrete realization of the general concept of the algorithm, and inherits much of its structure from that source. It is a strange and frustrating experience to read such books as an expert in computing. By the term "expert", I mean one

who both understands abstract algorithmic concepts and demonstrates effective programming skill. The expert wants a concept explained in general terms. A book almost always explains only by specific examples in its focus language. Of course, hundreds of authors are unlikely to be wrong. They use this concrete style of writing because it works to teach programming as it is usually taught in schools – as an isolated skill. It does presume that people learn computing ideas at first almost exclusively by learning a language. Skills-based learning makes it doubly hard to simultaneously master difficult abstract ideas. The tragedy in this all-too-common structure is that, without dedicated and formal study, it is hard to rise above the particulars of a language to see the general, powerful concepts at play.

This book is not an introduction to computer programming. Hundreds of books exist, and dozens of languages have new books published every year. It aims rather to help the amateur (and often self taught) designer/programmer become better at combining parametric modeling and programming to do more effective design work. This chapter presents a brief sketch of the sequence of concepts typically encountered in a programming language and outlines what each step in the sequence enables the designer/programmer to accomplish. For the novice, it might provide an principled overview, so often missing in basic programming books. For the expert, its brevity might help in reviewing and connecting key ideas.

4.1 Values

A *value* is a piece of data. Values are the basic objects over which computations occur. In computing in general, a value can be any *symbol.* Practically though, values usually come in kinds (or types, see Section 4.7 below). Most computer languages support a suite of such kinds, for example:

`5`	an integer
`3.14159`	a real number
`"f"`	a character
`"aalto"`	a string
`false`	a *Boolean,* either true or false

4.2 Variables

A *variable* is a container that holds a *value.* It has a *name.* We use variables to hold onto data so that we can use the data later. The nodes in a parametric model are, in fact, variables. They hold values, often multiple values (as shown in Section 4.8 below). In the following list, the variable name comes first, then the value it contains, then the value's kind of object. Variables names can (and often should) be long. A good (and common) programming convention, called *camel casing,* or *camelCasing,* makes memorable names by adding several words

together with no spaces and by capitalizing the first character of each word. The word "camel" refers to the "humps" in the variable names so created.

```
                    a=5   an integer
             SIRatio=1.414   a real number
buildingPart = "elevator"   a string
             qua = true   a Boolean
```

Variables enable *description.* With variables alone, a collection of data can be organized so that it makes sense to an external reader. Variables allow us to express a design as a collection of values.

By themselves, variables impose no ordering. A collection of variables, with no duplicates, can be considered in any order without changing the values held in the variables.

4.3 Expressions

An *expression* combines values, variables, *operators* and *function calls* that return values. Expressions are classified by the kind of value they return. For example, a Boolean expression returns a bit (either *true* or *false*). Expressions are units from which larger structures are built. Some examples of expressions include the following:

```
                a   a variable is a simple expression
        2 + (5 * 8)   this arithmetic expression returns the value 42
(1 + sqrt(5))/2   expressions can contain function calls
            b + 1   the variable b must be defined already
```

Expressions support *data dependency.* By using an expression, a piece of data can be computed from (made dependent upon) other pieces of data.

The values returned by expressions can be held by variables. This simple fact imposes a partial ordering on a set of variables. If an expression uses a variable, then that variable needs to be given a value *before* the expression occurs.

4.4 Statements

A *statement* is a unit of code that a language can execute. A program comprises a sequence of statements that are to be executed in the *given order.*

Statements can be *simple* (made of one statement) or *compound* (being a *block* made of a sequence of statements). Like a variable, a compound statement can be thought of as a container. In this case the container holds code.

A particularly important statement is the *assignment statement*, which *assigns* a value or the result of evaluating an expression to a variable. For example, the variable a may be assigned the value of the golden ratio.

```
a = (1.0 + sqrt(5))/2.0;
```

Two successive assignment statements with the same variable being assigned have the result of the second statement overriding the first. The statements

```
a = (1.0 + sqrt(5))/2.0;
a = 3.14159;
```

result in the variable a holding an approximation of the value of π, not ϕ the golden ratio.

Statements in a sequence, executed statement-by-statement, in the order given, capture key aspects of algorithms: ... *governed by precise instructions, moving in discrete steps* ... (remember Berlinski (1999)?). Together, variables, expressions and statement sequences enable a simple but useful form of algorithm.

Each parametric model node can be thought of as a program, that is, a sequence of statements. Each use of a value or of a constraint expression in a node property is equivalent to an assignment statement. Taken together, all of the individual node programs compose a larger program in which nodes must occur before they are used in another node's constraint expressions.

4.5 Control statements

The *flow of control* of a program is the sequence of statements that are executed when the program is run. Programming languages provide a class of statements whose purpose is to change the flow of control. The most simple of these is the if statement, which executes a block of code if some condition is true. Another example is the switch statement that provides a list of possible actions given the value of a variable.

```
if (a > 5.0)
  {
    // A sequence of statements goes here.
  }
```

4.1: When executed, an if-statement transfers the flow of control into its code block *if* the Boolean expression comprising its condition is true.

More complex is the *for-loop*, which calls an *initializer* for a *control variable* and repeats a block of code until its *loop condition* fails. At the end of each stage the for-loop executes a *counting statement* (the control variable usually occurs in this statement – exceptions abound, but these are hackery). Typical programming languages provide several such statements, for example, foreach and while.

```
for (i=0; i<10; i=i+1)
//for (initializer; loop condition; counting statement)
  {
    // A sequence of statements goes here.
    // Usually, but not necessarily, these statements
    // use the variable i, so that each iteration
    // through the loop has a different effect.
  }
```

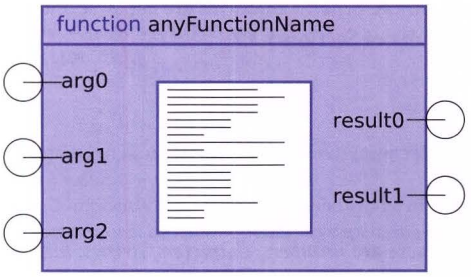
4.2: When executed, the for-loop initializes the control variable i and transfers control into the body of the loop. When it exits the body, it executes the counting expression, in this case it adds 1 to i. Then it tests the loop condition, in this case that i is less than 10. If true, it enters the loop body again.

Control statements enable programs to perform actions that depend on the *state of the program*, that is, the current variable assignments. Statements such as the for-loop facilitate expressing repeated similar actions as a single block of code, and thus can make programs much shorter, easier to maintain and sometimes more readable.

A simple list of statements can achieve a surprising amount. Just as in the real world where actions may depend on context, in programs, the computation may rely on data values. In the absence of control statements, programmers would have to write separate code for each situation and themselves determine which piece of code to use.

4.6 Functions

A function is a named block of statements, wrapped up in a box. See Figure 4.3. It has inputs (these are called *arguments*) that go into the box and returns values that leave the box. The code in the function acts on the inputs to compute the return values.



4.3: A function can be imagined as a box. The function is known by its name, in this case anyFunctionName. The arguments go into the left side of the box. The statements in the box use the arguments. The results exit the right side of the box as a result of executing the function. Unless there are side effects, the contents of the box are hidden from the containing program.

"Pure" functions have no effect other than to compute return values, that is, all of their action lies inside the box containing the function. Most programming languages though provide devices by which functions can have *side effects.* A side effect happens when a function either refers to or changes data outside its box. The most glaring example of a side effect happens when a function changes a *global variable,* that is, a variable that exists within the program as a whole and can be changed at any time by any function. As programs grow in size, side effects make them harder to understand and debug. Professional programmers will go to great lengths to avoid (or at least constrain) the use of global variables specifically and side effects generally.

Functions enable code reuse. Once defined, a function can be called throughout a program. The function itself is only stored once, reducing the length of code and, crucially, providing a single place for editing. To the rest of the program, a function is known only through its name and list of arguments. Programmers can, and do, change the code inside the function (to correct or improve it) and the rest of the program is not affected at all. This isolation of code by using an interface (the function arguments) is a simple form of what computer scientists call *encapsulation.*

Function calls enter programs as part of expressions. Every language includes a suite of predefined functions; programmers can define functions extending this set. Functions can call functions, enabling *composition* as a design tool for programming. Devising and refining layers of functions, each performing more specific work, is a key part of writing effective programs.

Functions are the first and simplest tool of *software engineering,* the discipline of building complex, reliable, maintainable and understandable programs.

In parametric modeling, a node can be thought of as a function call, in which an update method maps the inputs of the node to its outputs. Nodes can be drawn, and in some systems are so drawn, as function boxes with the arguments on one side and the returns on the other. Functions are the first and most simple device for building the modules of Section 3.3.7.

4.7 Types

Values have kinds. There are *numbers, characters, strings, bits* and others. *Types* organize these kinds by providing templates for their data and operators and functions that work with these templates. For example, the type `integer` gives a way to store integer numbers. It also provides such operators as `+`, `−`, `*`, `/*`, `<`, `<=`, `>`, `>=` and `==`; as well as functions such as `max(a,b)` (the maximum of two integers) and `print(a)` (returns true or false and has the side effect of printing the integer named by `a` onto the screen). User-defined types can extend the range of templates available.

Typically, variables must be *explicitly declared* to be of a specific type (although some languages have a generic type that holds any value). Expressions usually require their operands and function calls their arguments to be of specific types.

```
double a,b,c;
Vector p,q,r;
// initializing a, b, p and q
// initialization code goes here. examples below
a = 3.141592654;
b = 2.718281828;
c = 1.618033989;
p.X = 1.0;
p.Y = 2.3;
p.Z = 1.5;
...
```

(a)

```
// statements with expressions using the + operator
// integer example
int b = 4;
int c = 5;
c = a + b;          // c is equal to 9

// string example
string a = ''four'';
string b = ''five'';
c = a + b;          // c is equal to ''fourfive''
```

(b)

```
function CrossProduct(Vector xVec, Vector yVec)
  {
    Vector zVec; // declaration of return value type
    // code to produce zVec, the result
    return zVec;
  }
```

(c)

```
// a call to the function CrossProduct
r = CrossProduct(p, q);
```

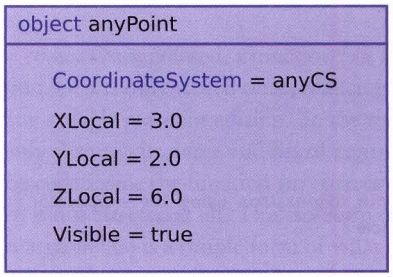
(d)

4.4: (a) Variables are declared to be of specific types. Programmers typically initialize them to a specific known value.
(b) The expression `a + b` implicitly requires its arguments to be of type integer, double or string. Operators that perform different operations depending on their input types are said to be *overloaded*.
(c) The function CrossProduct requires two objects of type Vector as input and returns an object of type Vector as output.
(d) Functions expose these type constraints when defined through their formal argument lists, but not in their actual argument lists when they are called.

Types enable a language compiler to perform some consistency checks prior to running a program, making some kinds of errors easier to find. Using explicit types helps as programs grow in size, but can hinder quick, exploratory coding.

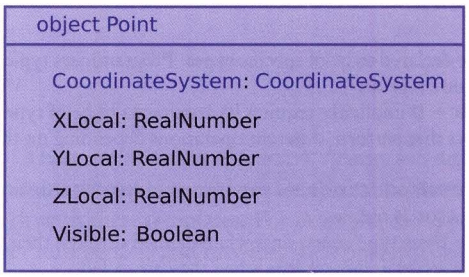
4.8 Objects, classes and methods

Objects generalize values. Whereas a value typically has little or no internal structure (an integer is just a number), an object combines multiple values (or other objects) into a coherent collection. Objects have *properties* (sometimes called *slots*), that is, named parts. *Dot notation* accesses these properties. If P is a point, the object P.X refers to the property holding its X-coordinate. Like a function, an object is a container; it contains values and other objects, not code.



4.5: An object can be imagined as a box. The object has a name, in this case anyPoint. The object's *properties* each have a name and a value. The values need not be primitive; they can be other objects, in this case, the CoordinateSystem property holds the value anyCS, itself an object.

Classes generalize types. A class is a template giving properties for objects of its class. An object can be an instance of a class – instances have all the properties specified in the class. The name of the class is its type. Each class property is of a particular type; only objects that are instances of that specified type can be held by the property. Classes are typically defined in *inheritance hierarchies* in which a class lower in the hierarchy has all of the properties of its parent(s) and may add additional properties. Almost all languages support *single inheritance*, in which a class may have only only one parent class. Mathematical and logical constraints make *multiple inheritance* difficult to include in languages and to use in programming practice.



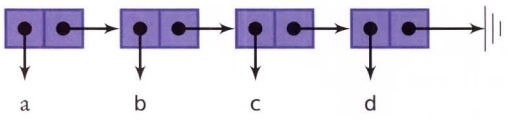
4.6: A class too can be imagined as a box. The class has a name, in this case Point. The class's *properties* each have a name and a type. The types need not be primitive; they can be other classes, in this case, the CoordinateSystem property holds a value of the CoordinateSystem class.

Methods are essentially functions specific to a class. Many methods of the same name may exist in a class hierarchy, each defines a function. The call *signature*, that is the types of the arguments in the method call, determine which of these functions is actually called. Such methods are said to be *polymorphic* (meaning of multiple bodies). Polymorphism enables programmers to express similar operations with the same name, thus potentially simplifying code. The same dot notation used for properties applies to methods. If P is a point, the method call P.subtract(Q) returns the vector that is the result of subtracting Q from P. Dot notation for methods makes object properties and methods almost the same from a programming perspective. It aids encapsulation – a programmer using it need hardly be aware of the internal structure of an object, only of the set of methods defined over the object.

Objects, classes and methods present a double-edged sword to amateurs. Largely, they help programmers make big programs more robust and understandable. Used well, they can make programs truly beautiful (at least to the eyes of us nerdy programmers) – they are elegant and powerful programming tools. Most modern languages implement some aspects of objects, classes and methods. Like power tools, they require setup and this takes time and effort. In the contingent, rushed style of programming usual in parametric design, minimal classes, simple objects and "messy" functions often produce acceptable results.

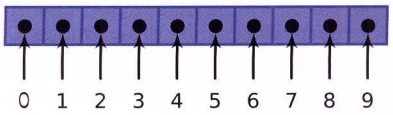
4.9 Data structures, especially lists and arrays

Data structures allow programmers to organize data themselves. A data structure comprises types (or classes) and functions (or methods) that perform coherent operations on objects of these types (classes). The *linked list* (see Figure 4.7) gives a basic example in which its single type has two properties, one for the first element (the *head*) of the list it represents and the other for the rest (the *tail*) of the list. To access a member of a list, one must start with the head of the list. If the head is not the member sought, visit the tail, until the desired member is found.



4.7: The linked list data structure comprises a single class (or type) often called List, ListElement or Cons containing two properties. The first property (called Head, First or Car) points to a value held at the place of the cell in the list. The second property (called Tail, Rest or Cdr), points to the rest of the list from that cell. The symbol ||| refers to the null list, and is often called nil. Instances of the List type are organized to represent structures including lists, trees, directed acyclic graphs and general networks.

Like a linked list, an *array* implements a sequence of objects. Unlike linked lists, access is by *index,* that is, position in the collection. Typically, array positions start at `0`, so the expression `a[0]` gives the first element of the array and `a[2]` yields the third element.



4.8: The array data structure comprises an ordered collection of cells and an associated *index set.* The cells hold data. By common convention the index set comprises the natural numbers, that is $0, 1, 2, 3, \ldots$. A member of the index set accesses the associated cell, for instance, 4 accesses the fifth member of the array. In most programming languages index sets start at zero, which creates linguistic, but not mathematical, difficulty in working with arrays. This quirk is a fact of history and programmers just have to get used to it.

Data structures are a key abstraction technique in programming. Once built, they can be used over and over again without worrying about how they do their job. Lists are just about the simplest data structure. They are easy to use, have a huge range of operations and functions and can hold values of any type. On the other hand, for some operations they are not very efficient and their very generality can make them hard to debug and maintain. Lists and arrays are well suited to quick, contingent programming in parametric modeling and are the first structure that modelers should learn to use and make.

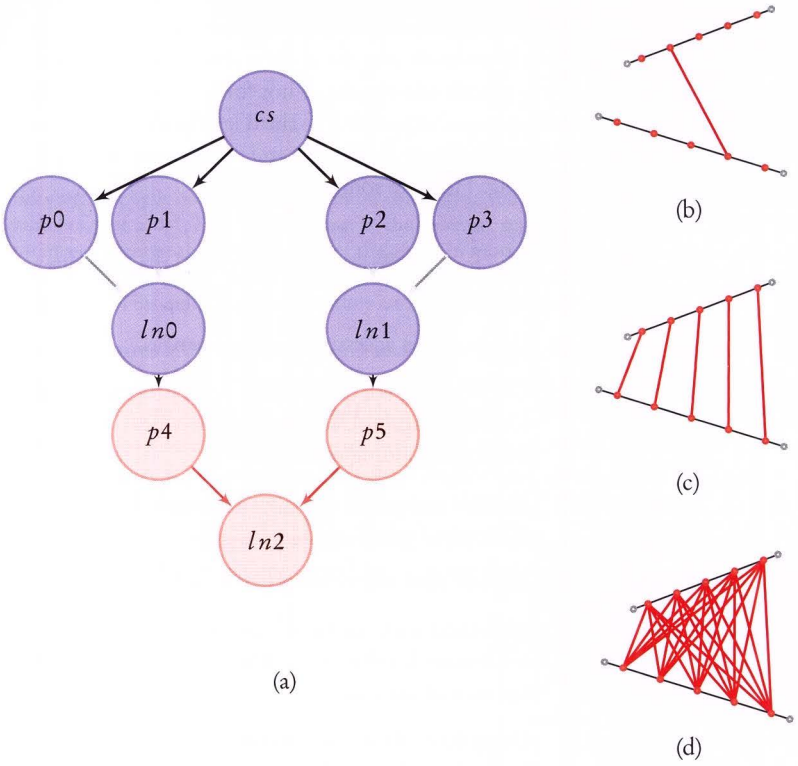
4.10 Conventions for this book

Snippets of code appear throughout this book. The "language" to use for these was a difficult decision. The choices were three: an existing language, faithfully reproducing its syntax; the *pseudo code* used by computer scientists to express algorithms for publication; or a simplified language hopefully readable for many. I rejected the first two choices. The first requires the reader to know a specific language and might give the impression that the book is somehow about that language. The second, while precise and elegant, is not for amateurs, who often have not bridged to the abstractions required. The third lent itself to the license I needed, both to express ideas as simply as possible and to add some notation specific to parametric modeling. Here are some conventions:

`// comment`	Two forward slashes turn the rest of the line into a comment.
`p.X`	Dot notation accesses object properties.
`CamelCasing`	Camel casing is a convention, not a programming language feature. Used to combine words into names while maintaining readability.
`variableName`	Variable names begin with a lower-case letter and are otherwise CamelCased.
`TypeOrClassName`	Type or class names are pure CamelCase.
`Point p = new Point();`	A method with the same name as a class defines a *constructor* for that class. When called, it produces an instance of the class.
`p.ByCoordinates(1,4,3)`	Using `By` or `At` in dot notation signals that the method is a node update method.
`a = {1,3,6,3,8}`	The principle of *replication* is that a variable can hold a list or a single value. When a function is called on a list it applies to each element of the list in turn.

Replication (Aish and Woodbury, 2005) needs some explanation. A node's independent variables may be either *singletons* or *collections*. A collection has the interpretation that each object in the collection specifies a node in and of itself. When multiple variables representing independent nodes are collection-valued, collections propagate to variables representing dependent nodes in two distinct ways as shown in Figure 4.9. The first produces a collection of objects, of size equal to the shortest of the input collections, by using the i^{th} value of each of the input collections as independent inputs. The second form generates the Cartesian product of the input collections. The Cartesian product $X \times Y$ of collections X and Y is the collection of all possible ordered pairs, taking the first member the pair from X and the second member from Y. For example, $\langle 1,2\rangle \times \langle a,b,c\rangle = \{\langle 1,a\rangle, \langle 1,b\rangle, \langle 1,c\rangle, \langle 2,a\rangle, \langle 2,b\rangle, \langle 2,c\rangle\}$. Both cases result in

a single node in the graph, with its elements being accessed through an array-indexing convention. The identification of singletons and collections supports a form of programming-by-example whereby the work done to create a single instance can be propagated to multiple instances simply by providing additional input arguments.



4.9: A line joining two point collections, each in turn expressed as a parametric point on a line. (a) A symbolic model representing the parametric points and connecting line. The same symbolic model represents (b), (c) and (d). The diagram (b) shows a line between the two parametric points with single index values (1 & 3) for the parameters of its defining points. (c) Each of the defining points of the result line uses all of the values in its input collections. The number of lines is equal to the length of the shortest collection. (d) A line collection under the Cartesian product interpretation of a collection joins each point in the first to each point in the second collection.

4.11 It's more than writing code

Programmers use the above language constructs (and others) to write programs. The act of programming itself has several facets.

To *design* code is to understand the problem, decompose the problem into parts, devise data structures and algorithms for the parts and compose the parts into an entire program design.

Coding translates a design into a program. It takes the abstract ideas of a design and turns them into precise instructions in some programming language. Code seldom works as written. Sometimes, coding and design go together, especially at early, exploratory stages of an idea.

Errors, which programmers call *bugs*, make themselves evident anytime from initial compilation to only after several years of use of a particular piece of code. Finding and fixing such bugs can be a fascinating intellectual activity in its own right. Without any slight to programmers, it is common that more time goes to *debugging* than any other part of the programming task. In fact, programmers will express real surprise when a piece of code works the first time as written!

A program may work, but may be unclear or may need to be used in a more general way. *Refactoring* is the process of *re-designing* code to improve its clarity and its interfaces to other code. Refactoring makes code more adaptable so that it can work in a range of situations.

Most good, large programs are built in *modules.* A module is a collection of data structures that implement a coherent and consistent behaviour. For example, in geometric computing, a common low-level module implements a concrete form of *vector spaces,* that is, collections of vectors that obey certain mathematical rules. Vector spaces rely on real number arithmetic in the language below and provide consistent vector operations to programming layers above. Vector space operations do not include any concept of location, as vectors are simply directions and magnitudes. Location is typically introduced in an *affine space* layer above vector spaces. To design and program in modules is to conceive of the "world" being programmed as having multiple descriptions, each one in turn expressed as a description in some more atomic module. Designing and implementing systems made of modules is the focus of the discipline of software engineering.

A very important and surprisingly difficult programming skill is to abstract to the lowest level. If an operation or piece of data can be expressed without a domain-specific term, it should be. For example, inserting punched windows and doors into walls can be accomplished by devising a data structure specific to walls, into which holes are cut by special-purpose functions. More abstractly, walls can be represented as solids, in which case the hole-cutting operation can be conceived as the subtraction of a sweep of an outline representing a window or door from a solid representing a wall. In the latter case, details of the wall (its construction, thickness and shape,...) are invisible to the solid representing it. In turn, the hole-punching operation relies only upon the geometry of the solid.

Functions and data structures can be general or specific, complete or partial. To be general is to accommodate many cases. To be complete is to handle all cases in some logical class. For example, data structures and functions over vectors will be general as vectors are the basis for almost all computational geometry. A complete set of functions for vectors might take a very long time to write.

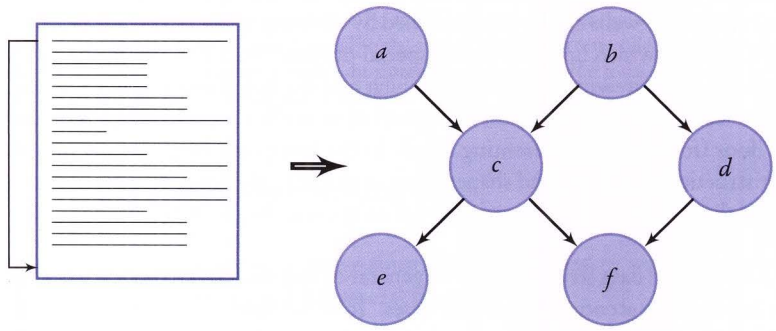
Most languages come with associated data structures and functions. These are almost always incomplete in some context and for some tasks. Programmers must write their own functions when needed. People who spend a lot of time programming will often build up personal collections of classes and functions, which they use and refine again and again in new projects.

Programming is algorithmic thinking in action. Two programs may express an identical algorithm, yet differ in fundamental ways. Above the basics lies a craft of programming, which takes time to master. The craft comprises concepts, constructs and skills. Parametric modelers are mostly amateur programmers. Their work patterns show a tendency to short code in which the craft manifests to a greater or lesser degree.

4.12 Combining parametric and algorithmic thinking

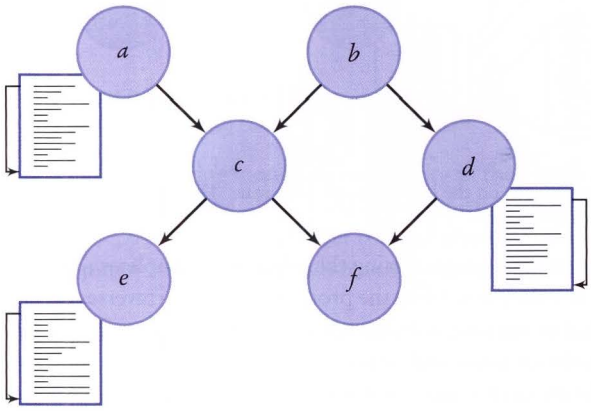
Programming enters parametric modeling in four distinct ways: parametric model construction, update method programming, module development and meta-programming.

Almost all conventional CAD systems have a programming language, either internal or accessible from the system. Designers program in these languages to build and edit models. Once built, models can be changed either by hand or by the action of other programs. Certainly, parametric thinking can and does engage programming of this sort. Programmers use some of the variables that are passed to functions as parameters that link to new parametric structures created in the program. An early CAD book, *The Art of Computer Graphics Programming: A Structured Introduction for Architects and Designers* (Mitchell et al., 1987) was essentially the reverse of this book. Through many examples, it showed how to build a parametric layer onto the top of a structured program. Parametric modeling inherits this programming mode but builds parametric models, that is propagation graphs, rather than fixed models.



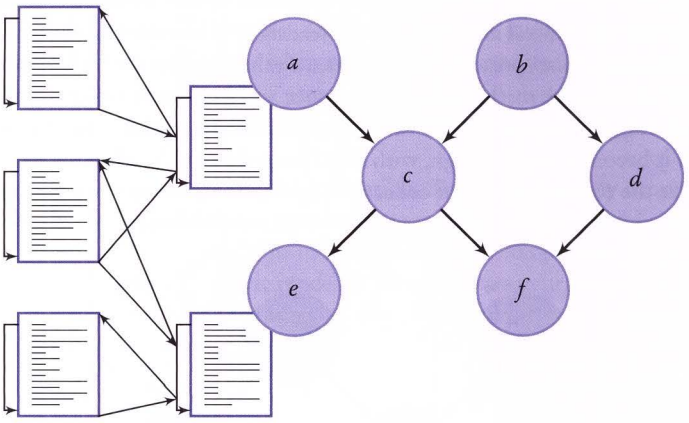
4.10: When executed, the block of code on the left creates or modifies the parametric model on the right. The arrow along the side of the code block represents the flow of control through the code when it is executed.

The second role of programming is in writing node update methods. This is like writing expressions in the cells of a spreadsheet. The expressions are called at each update of the spreadsheet to produce a value. Unlike the formulae in spreadsheets, update methods are written once and used many times by calling, not copying, the method. In this role, programs may be spread around a model, so that it becomes hard (though, with small programs, this is seldom needed) to visualize the code as a coherent collection. In this mode of work, each program stands on its own, at most calling other functions defined elsewhere.



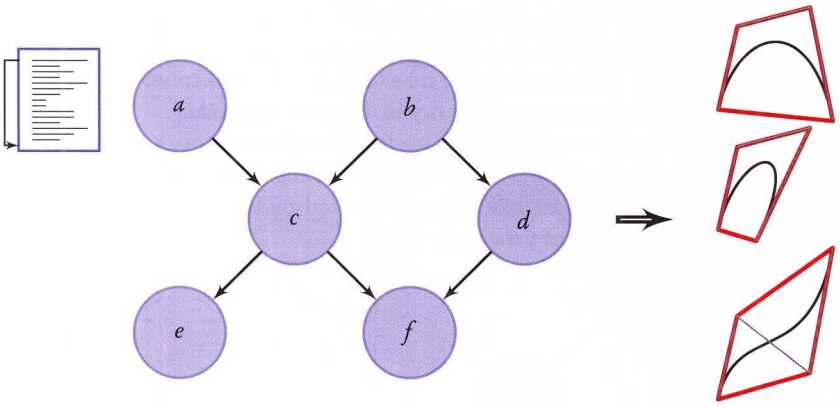
4.11: The nodes a, d and e have user-defined update methods. As propagation visits each model node, for example, in order $\langle b,d,a,c,f,e\rangle$, it executes the node update methods for each node in turn.

Creating a module requires design, coding, debugging, refining and maintaining a data structure and a suite of functions over that structure. Such new modules are needed if a system does not support a particular design task. For example, a layout module for rectangular rooms (see Figures 3.8 and 3.10) requires data structures representing rooms and walls, and functions for inserting, removing and dimensioning both. It can take a great deal of time to build such structures. Consequently, complete modules are a relative rarity in designer-built code. Once a serious commitment to parametric craft is made (or sneaks up through sustained work), it is inevitable that a designer/programmer will build her own modules. As with the master carpenter's jigs, clamps, racks and guides, these modules become an integral part of the parametric craft.



4.12: The code blocks on the left represent functions. They are called by the update methods on nodes *a* and *e*.

Meta-programming – programming the program – completes the set. It affects the whole, not the parts. Here the program affects or traverses the propagation graph itself. For instance, a design space explorer program takes a model and a small set of source nodes and systematically tries combinations of node values, updating the model for each combination and reporting the results, either on the screen, to files on the computer or to another process. Systems enabling meta-programming provide a set of functions that control *graph updates*. When called, these functions invoke the graph propagation algorithm starting either at the sources or at specified nodes. Graph updating provides a key entry point for techniques that can make a propagation-based parametric modeler perform cyclical calculations, perform systematic searches and produce animations.



4.13: The code block on the left acts as a meta-program. It resets some of the graph-independent properties of the model, calls `UpdateGraph()` and records the result external to the system.

Building and using parametric models mixes propagation graphs and programs. Deciding when and how to employ each mode of work is itself part of the craft. Sometimes, clarity comes from the careful construction of new parts from old by building successive nodes in the graph. For instance, the two-circle tangent construction in Figure 2.17 builds and demonstrates a clear geometric method. On the other hand, a formula may be at hand that reduces this construction to a simple set of assignment statements, which is then wrapped in a function. Proficient parametric modelers routinely slip from modeling to programming and back.

4.13 End-user programming

Designers are not alone in facing increasing complexity in their tasks and tools. Many disciplines face a fundamental need and opportunity to do more with their computing tools. All encounter the fact that the graphical user interface, which makes computers so easy to use, also makes them hard to use powerfully.

The graphical user interface (GUI) has profoundly changed our engagement with computers. It does so by providing a shared visual metaphor that enables manual interactive tasks. It largely ignores computation's most vital aspect – the algorithm. Far too often, people must perform repetitive tasks through the GUI that could be completed more quickly and correctly with an algorithm. End-user programming tools promise to support people in expressing and using algorithms within computing tools such as spreadsheets, word processing tools, image systems and computer-aided design systems. However, useful end-user programming systems have been hard to achieve.

End-user programming systems aim to amplify work. They support domain specialists in doing work "better" (this means being more effective, efficient or replacing old tasks with new tasks). End users program to resolve unusual or repeated tasks. Their knowledge and skill lie within their domain and they have acquired programming ability as an adjunct. Further, they view their work as being primarily in their domain, rather than as the development of programs to support others (though many end-user programs are used by others). The point here is that the task comes first and programming is a means to its end.

Typically, end-user programmers work with specialized software. Writers use Emacs, Microsoft Word® or the Adobe Creative Suite®. Designers may use ArchiCAD®, AutoCAD®, CATIA®, form•Z®, GenerativeComponents®, Maya®, Revit®, Rhinoceros®, its add-on Grasshopper™ or SolidWorks®. Game designers may use Cinema4D® or Virtools®. They program if the task at hand requires much repetitive work, involves redundant data or must cohere in some way; and when the tools available make work difficult to accomplish. Motivation increases for unique, high-value tasks; when the work repeats; or programs will be reused.

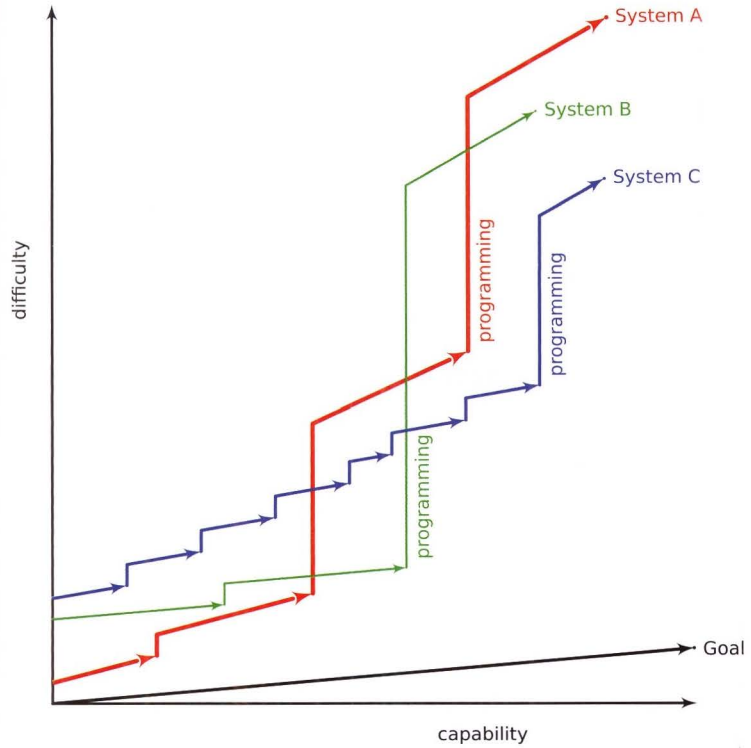
End-user programmers "come out" of their own domain to clarify, abstract and generalize. End-user programming is thus a form of meta-work, in which the programmer must reflect on the tasks at hand, develop, test and refine tools to aid it and then use those tools to accomplish the actual tasks.

End-user programming comes with costs. Increasing capability adds complexity. First introduced by Dertouzos et al. (1992) as *gentle-slope systems* and further developed by Myers et al. (2000), each end-user programming system has an informal function showing how difficulty increases with capability. Systems typically display steps in these functions that correspond to the need to learn new programming constructs and ideas. Figure 4.14 shows the ideal of slowly increasing difficulty with capability; the typical situation in which difficulty becomes an insurmountable obstacle to progress; and a realistic goal in which simple programming features can be learned and used incrementally without removing the end-user programmer from his task.

Parametric modelers do have common cause with professional programmers. Software engineering is the body of knowledge and craft for making provable, reliable, reusable and maintainable programs. In recent years software engineers have paid considerable attention to so-called *agile methods* (Highsmith, 2002), in which programs and their specifications develop in tandem, and programmers work in continuous consultation with those who use (or will use) their work. The *Manifesto for Agile Software Development* (Beck et al., 2009) declares four core principles for agile methods:

> Individuals and interactions over processes and tools
>
> Working software over comprehensive documentation
>
> Customer collaboration over contract negotiation
>
> Responding to change over following a plan
>
> That is, while there is value in the items on the right,
> we value the items on the left more.

These sound much like design. Clearly, the contingent, task-focused style of parametric work and agile methods share much common ground. At the time of writing though, there was little explicit connection between them.



4.14: A good strategy for end-user programming is to aim at a relatively large number of small steps in the capability/difficulty function. Adopted from Myers et al. (2000) the diagram compares fictional but representative systems offering similar capabilities. The intent of the figure is to give a sense for the relative difficulty of using tools to achieve results. The specific shape of each curve can be only anecdotally explained. For instance, System A provides an initial low barrier to use, but has a confusing interface that makes it difficult to learn new features. It presents a significant incremental step when its key feature of constraints are used. Its scripting language is separate from the interface and is based on an old and inelegant language, making it hard to connect programs to the model. In comparison, System B's user interface has a high initial barrier as it is principle-based, but a low slope as its principles make it predictable. Programming in System B requires invocation of an integrated development environment, and this a barrier to its use. In System C, programming language constructs are available directly in the interface and are carefully factored so that they can be largely used independently. A typical end-user programer learns and uses these features in increments, seldom straying far from the task at hand. At some point, the end-user programming does need to take advantage of the full system capabilities. The jump to its full programming environment is reduced by prior practice with programming elements. Though these three systems are abstract, their basic structure can be found in several CAD systems that were on the market in 2010.

Chapter 5

The New Elephant House

Copenhagen, Denmark
Architect: Foster + Partners

by Brady Peters

5.1 Introduction

Copenhagen's New Elephant House opened in June 2008 replacing a structure dating from 1914. The Copenhagen Zoo, set within a historic park, is one of the largest cultural institutions in Denmark. The New Elephant House seeks to create a close visual relationship between the zoo and the park, to provide the elephants with a stimulating environment, and to create exciting spaces that provide excellent views of the elephants. The House brings a sense of light and openness to a traditionally closed building type. Two lightweight, glazed domes cover the building and maintain a strong visual connection to the sky and the changing patterns of daylight. The elephants can congregate under these glazed domes or out in the connecting paddocks. In the wild, bull elephants have a tendency to roam away from the main herd. The plan form therefore comprises two separate enclosures, a large one for the main herd, and a smaller one for the more aggressive bull elephants. Dug into the site, the building has minimal visual impact on the landscape and excellent passive thermal performance. For visitors, a ramped promenade leads down through the building looking into the elephant enclosures along the way.

This chapter focuses on the design of the glazed domes of the New Elephant House. The canopy design was explored in many ways, through sketching, physical model making, and three-dimensional explorations using computer modeling. The torus, a mathematical form, harnessed the complexity of the design by providing a geometric logic to which the structural and glazing could relate. A parametric computer model encoded this set-out and constructional logic. This allowed for the generation and exploration of many different design options. As the design comprised a collection of relationships and the computer

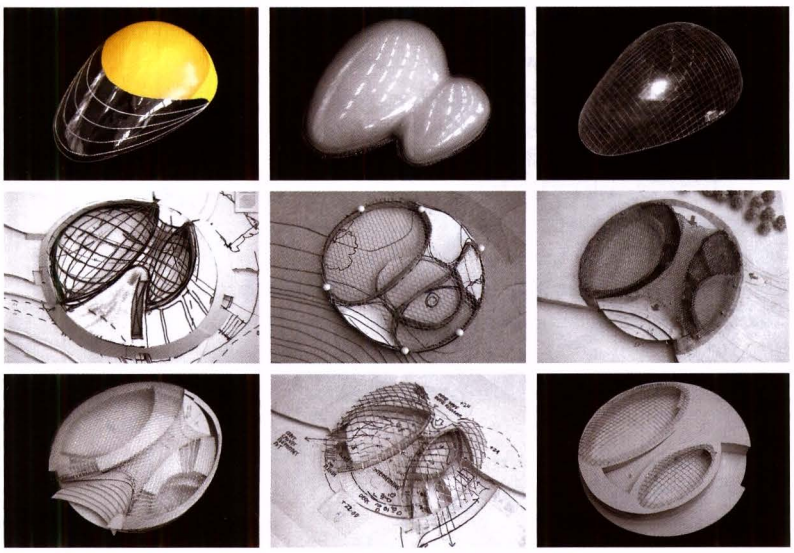
model could be updated instantaneously, the design remained fluid until late in the design process. A series of opening panels and a varying fritting pattern on the glazing panels of the canopy formed the design's environmental strategy. The design of this system – the distribution of the different panel types and the creation of the bespoke fritting patterns – was explored using custom computer programming. A design emerged that incorporated a semi-random placement of leaf textures. This created an environment with different light levels simulating the elephants' natural environs.

5.2 Capturing design intent

The architect's design studies suggested two canopy structures rising from the landscape, one larger than the other, with the bulk of the building built into the earth. The two canopies relate to the internal arrangement of the elephant spaces and to the landscape. The structure became an array of members defining quadrilaterals and covered by a fully glazed surface. Both canopies had double curvature and the glazing followed the quadrilateral structure geometry.

While these design studies used many media, physical model making played a crucial role. Both the architects and structural engineers made models to test spatial, formal and structural ideas allowing ideas of structure and of form to interweave. Specific model-making techniques and many different materials carry with them their own material logic. This inherent material logic can be explored to achieve families of similar options. Shown in Figure 5.1, canopy design concepts were developed and tested using many form-making techniques: grid shells made from wood and metal, form-found models in metal and fabric, sculpted vacuum-form models, cable net structures and bendable metal mesh each suggested exciting new formal compositions.

As the design rules developed, a more descriptive solution became necessary, and digital models became essential. The geometric logic of the New Elephant House was neither pre-rationalized nor post-rationalized; the construction system concepts developed with the design. When a design concept became both interesting and sufficiently clear, it was translated into a computer model. "Interpretation" is perhaps more apt than "translation". The digital medium usually suggests or even demands new geometric logic. The particulars of this new logic depend on the software tools and the skill of the person doing the modeling. In this project, one of the first tasks for computer modeling became templates for making detailed physical models of the canopy structure. Through sketch CAD models the design team resolved dimensional characteristics of the structure and its set-out. The geometric complexity of the canopies required exploring these digital sketches with three dimensional CAD models, not just simple two dimensional drawings. This was an important part of the design process, and computer modeling an essential tool throughout the project, not just a drafting and rationalization tool to be used at the end of the project.



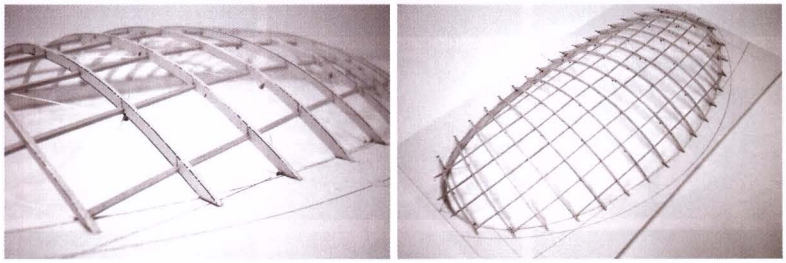
5.1: Physical sketch models.
Source: Foster + Partners / Buro Happold

5.3 The torus

The torus, colloquially known as a "donut", is a mathematically defined surface of revolution. It is generated by revolving a circle around an axis lying outside and in the plane of the circle. Useful defining parameters for this surface are the radius of the circle, and the distance of the circle from its rotational axis. The torus form has many benefits for architecture: the surface is constructed from a series of arcs; the arcs in the rotational direction are equal; the surface can be discretized into planar four-sided panels; those panels are identical when rotated about the torus's axis, but not along the defining circle; and the panels align with each other along their edges. The torus thus defines an array of planar faces suitable for manufacture. Project cost constraints put a high priority on using repeated identical panels. This geometric set-out is based on arcs, another very useful property as this allows for reliable solid and surface offsets, which helps to resolve many complex issues of design and production. Typically, a project uses only part of a torus surface, which is referred to as a *torus patch*.

Physical model making motivated the inital computer modeling. The structure was set-out on the computer, and then assembled by hand. While the torus is a very clear and practical form, it does not capture the playfulness that existed in many of the original physical study models. The early studies of the torus form produced canopies that did not relate well to each other or to the plan beneath. A more flexible form was needed. Figure 5.3 demonstrates the key discovery: tilting the axes of two torii. Assymetric forms result by slicing these tilted torii by a horizontal plane By tilting each torus in opposite directions, the plan form

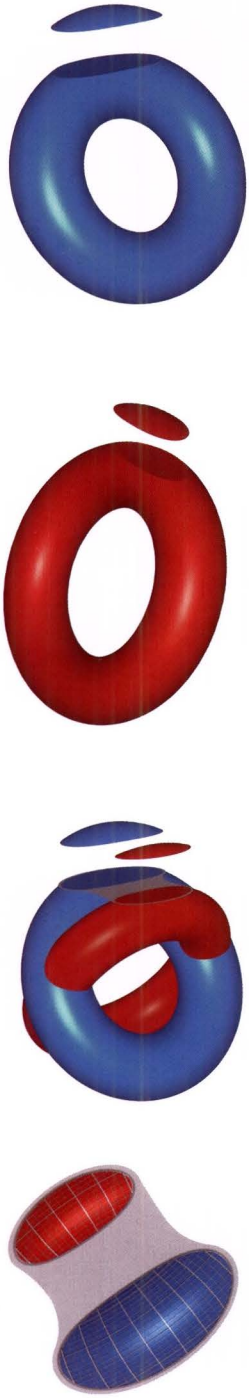
of both canopies defined a central area between the domes. By adjusting the parameters of the torus and angle of axis tilt, the form of the canopies came to both define and fit the New Elephant House plan.



5.2: Study model of torus geometry
Source: Brady Peters / Buro Happold

As shown in an initial sketch model in Figure 5.2, the set-out for the structural and glazing systems follows the torus geometry. Structural centrelines, as well as beam and glazing elements, derive from the torus geometry. The structure and glazing systems of the canopy terminate at a structural ring beam. This ring beam lies on the horizontal plane intersecting the torus.

To arrive at the basic spatial composition, the design team employed a variety of media in sometimes unanticipated ways. The team started with sketches and physical models, worked through a stage of literal computer modeling and then used parametric modeling to discover and refine a simple underlying geometry giving a complex visual form. Ironically, once discovered, the form's geometric simplicity meant that designers could choose either computational or analogue tools in further work.



5.3: Torus geometry set-out.
Source: Brady Peters,
based on Foster + Partners design

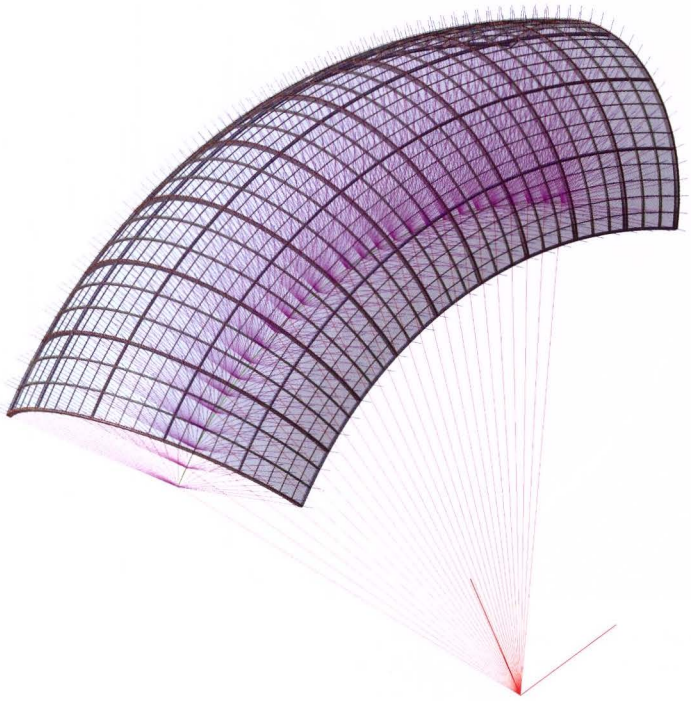
5.4 Structure generator

As with physical models, design ideas in digital models are often first developed in a manual fashion, however, as the geometric rules and construction details are established the case for investing in a parametric model grows.

With the two-torus geometric framework (though not its specific parameters) decided, the designers turned to structure and glazing. They quickly found their task to be designing a family of ideas and discovery of a specific solution, rather than simple detailing of a single sketch. The level of complexity and the sheer number of potential configurations necessitated a parametric approach. The team decided to work with an architectural designer possessing programming skills to write a custom program, called the *structure generator*. Programming freed the team from the limited command palette available in any particular CAD package. In use, it became like any other design tool – applied iteratively

throughout the process. Instead of drawing with a pen, the designers sketched with code.

The careful creation (and naming!) of appropriate variables determines much of usefulness of a parametric system. For the New Elephant House structure generator, 26 variables controlled the number of elements, the size, spacing and type of the structural members, the different structural offsets, the primary and secondary radii of the torus, and extent of the structure generated. In turn, these numeric variables related to the torii axes expressed as coordinate systems. The structure generator produced all of the centrelines, primary, secondary, tertiary, quaternary structural members, glazing components, as well as tables of node points.

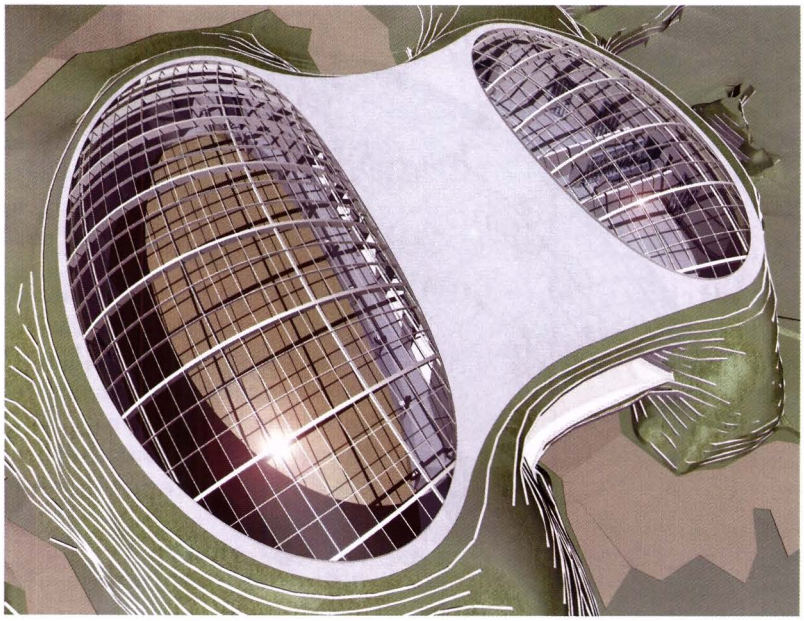


5.4: Structure generator interface and generated geometry.
Source: Brady Peters, based on Foster + Partners design

In this project phase, programming a parametric model enabled creating and testing many variations of structure and canopy within the two-torus form, itself a parametric model subject to change. Through the use of the structure generator many more options could be studied than if the canopies needed to be rebuilt with each new option. The speed of producing new options also allowed

the canopy design to be changed late in the design process. Here computation became a refinement and optimization tool, resulting in the design shown in Figure 5.5.

The fabricator received the dome designs as a document called the *Geometry Method Statement*, rather than through a computer program or digital model. This simple, verifiable document assures reliable data transfer between CAD systems – fabricators must build their own digital models following its rules. As an educational and contractual strategy the Geometry Method Statement helps fabricators fully understand the geometric complexities of the project. This document describes the design in terms of simple geometric rules. For the New Elephant House project, it follows directly from the set-out logic of the torii and the structure generator computer program.



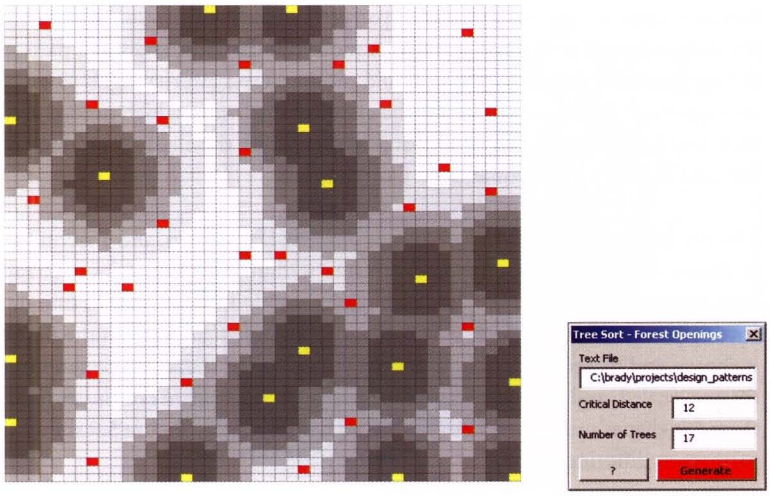
5.5: Elephant House canopy structure.
Source: Brady Peters, based on Foster + Partners design

5.5 Frit generator

The environmental strategy for the project was expressed both through a series of opening panels in the canopy, as well as a varying fritting pattern expressed on the glazing panels of the canopy. Through the use of a computer program, patterns emerged through a semi-random placement of leaf textures.

Environmental performance and occupant comfort were important design goals. The design team decided that glazing panels themselves should do as much environmental control work as possible. Through a series of opening panels in the canopy and a varying fritting panel it achieved ventilation, solar control and variable lighting simulating natural conditions. Variable openings in the glass canopy controlled natural airflow. Fritting patterns printed on the glass reduced solar radiation received and thus helped maintain a comfortable temperature. No other coatings were used on the glass so that the light within the elephant enclosures would be as natural as possible.

The solar control of the fritting depended on the local ratios of transparent to opaque areas. The environmental analysis defined the level of fritting and the number of panels of each type of fritting density. While the overall amount of fritting was critical, the distribution of these different panel types was not. A new distribution pattern for the different panel types was developed, dubbed the TREE SORT pattern and shown in Figure 5.6. As wild elephants gather at forest edges, the forest became a metaphor for distributing shading panels and fritting density. In the TREE SORT pattern, the opening panels are analogues of forest openings and therefore have no fritting. The pattern found a specified number of tree trunks (yellow panels) as far away from openings (red panels) as possible, and created a gradient of panel types radiating away from the tree trunk (See Figure 5.7). The dense areas of fritting centred around tree trunks, with decreasing density from the trunks towards the opening panels. The design team explored a range of results by adjusting the position of opening panels, the number of trees, the minimum distance between trees, the number of panel types and the number and distribution of each panel type.



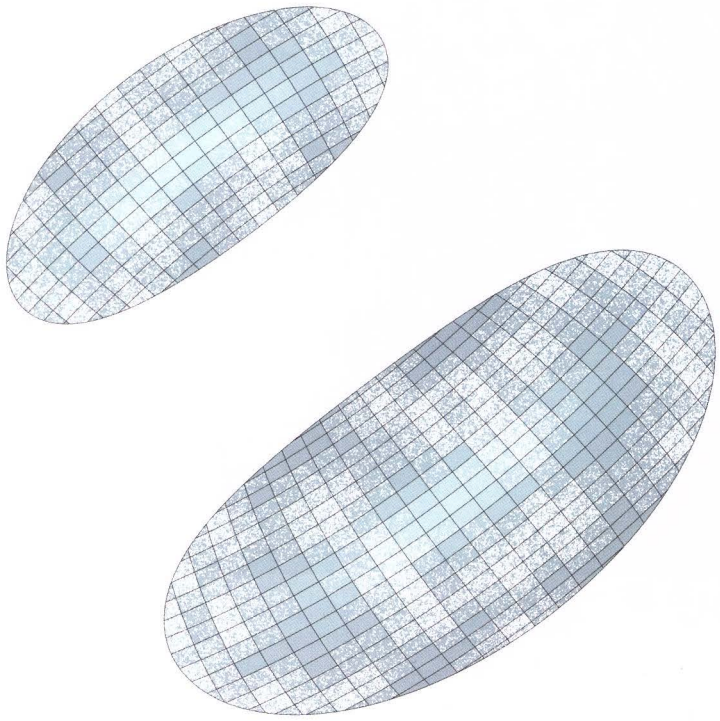
5.6: TREE SORT pattern
Source: Brady Peters

5.7: Panel Type Distribution on Canopies.
Source: Brady Peters, based on Foster + Partners design

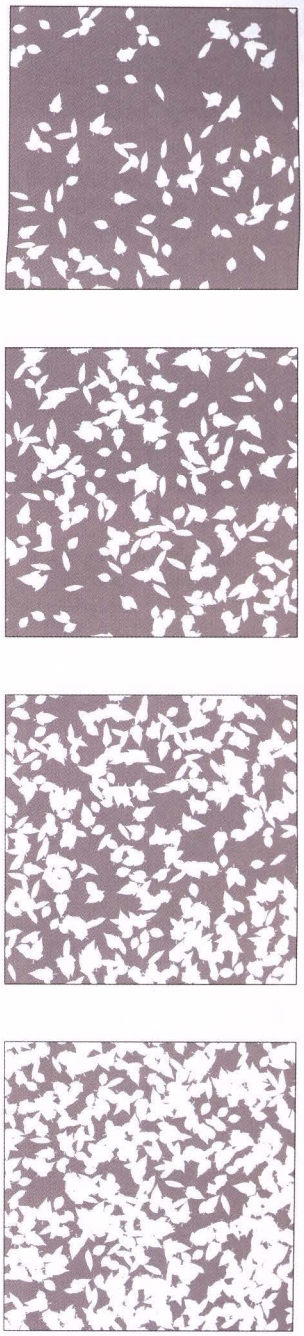
Another computer program, called the *frit generator* and shown in Figure 5.8 was developed to create a custom frit pattern for the New Elephant House. The frit design started with a leaf pattern. A more standard micro-dot frit pattern did not work for this project as it would produce even internal light, suitable for an art gallery or office, but not for the elephant enclosure which needed areas of light and dark contrast. The intent is that this allows the elephants to seek out the area in which they would most like to stand.

The frit generator takes a series of shapes for the frit pattern, and a second series of shapes that the frit pattern will be created within. This last series of shapes represents the glazing panels. For each panel, the algorithm creates a unique fritting pattern as shown in Figure 5.10. A random frit shape from the first set is chosen and placed within the glazing panel. The frit shape can be randomly rotated, scaled, and its vertices subtly shifted. The algorithm continues to place these frit shapes until it reaches the desired frit area ratio. Figure 5.9 shows frit patterns at differing ratios.

5.8: Frit generator with interface and generated frit shapes.
Source: Brady Peters, based on Foster + Partners design



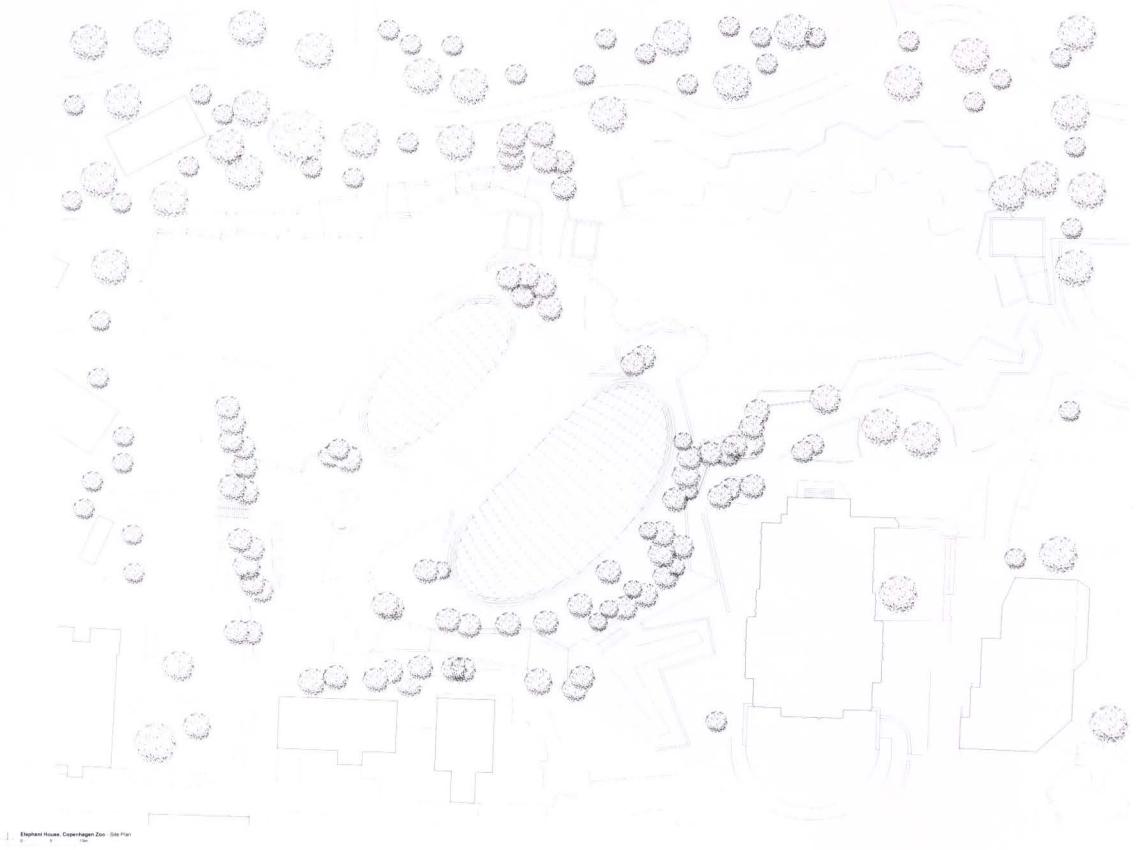
5.10: Frit patterns distributed on canopies.
Source: Brady Peters, based on Foster + Partners design



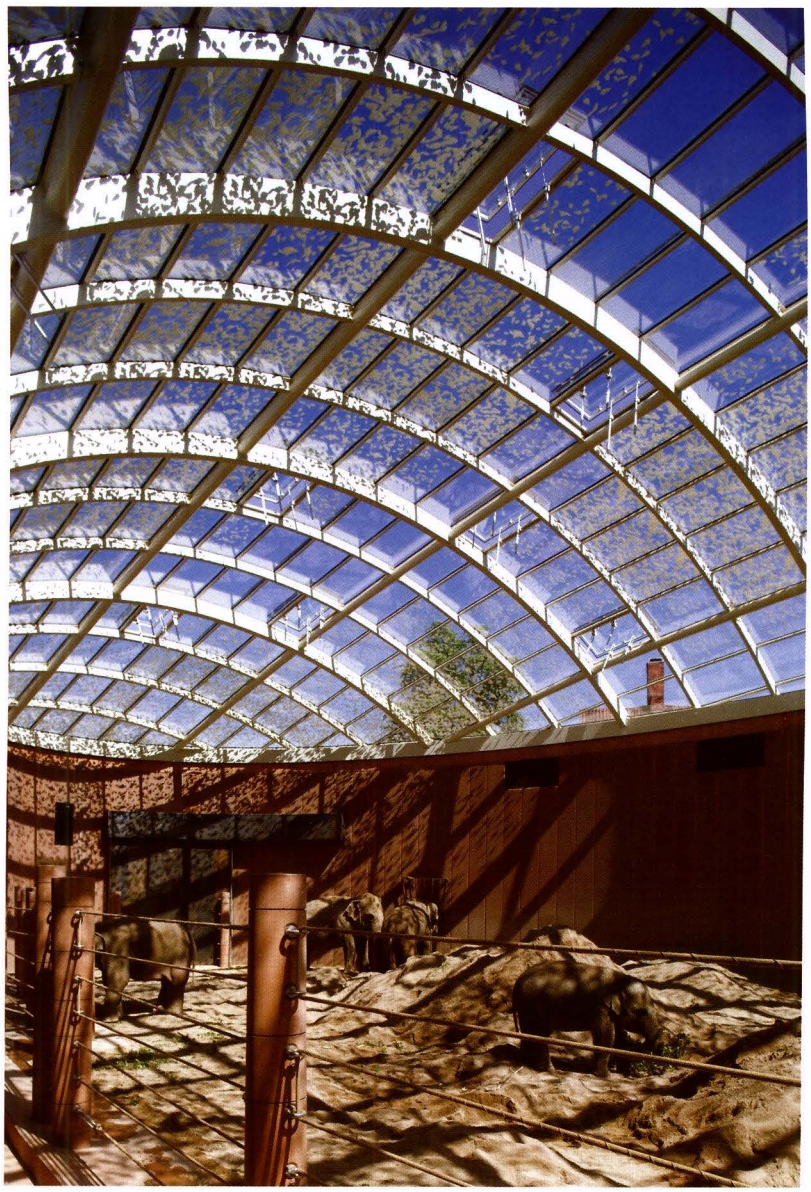
5.9: 15%, 30%, 45%, 60% Frit patterns.
Source: Brady Peters, based on Foster + Partners design

5.6 Conclusions

The program for the New Elephant House in Copenhagen had both restrictive constraints and a complex and untested set of requirements. Its design process used many different media, both analogue and computational. Both physical and digital model making contributed to the design outcome. The mathematical form of the torus helped to achieve both an economy and a constructional logic for the project. A custom computer program enabled extensive exploration of the three-dimensional geometry of the digital model. This generation method helped to optimize building form and structure. The project's environmental performance was integrated into the design through new panel distribution patterns and semi-random fritting patterns. Figure 5.12 shows that the project incorporates patterns from nature, patterns from geometry and patterns from computation.

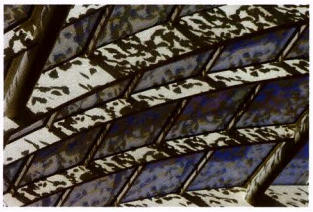


5.11: Plan of the New Elephant House.
Source: Foster + Partners



5.12: Interior of the main herd enclosure in the New Elephant House.
Source: Richard Davies / Foster + Partners

5.13: Detail of roof opening.
Source: Richard Davies / Foster + Partners



5.14: Detail of frit pattern.
Source: Richard Davies / Foster + Partners