FormWriter

A Little Programming Language for Generating Three-Dimensional Form Algorithmically

Mark D. Gross
Design Machine Group, Department of Architecture, University of Washington

Key words: Programming Language, Geometry, Form Generation

Abstract: FormWriter is a simple and powerful programming language for generating three-dimensional geometry, intended for architectural designers with little programming experience to be able to generate three dimensional forms algorithmically without writing complex code. FormWriter's main features include a unified coding and graphics environment providing immediate feedback and a "flying turtle" - a means of generating three dimensional data through differential geometry.

1. ALGORITHMIC FORM GENERATION

1.1 Introduction

Powerful three-dimensional modelling software has made it easy for architects to shape and sculpt three dimensional material and space using direct manipulation operations. Architects find that computer aided design modellers can help them generate radically new forms for buildings. Gehry's forms in the Guggenheim museum in Bilbao and at the Experience Music Project in Seattle, Greg Lynn's proposals for blob architecture (Lynn, 1998); Reiser+Umemoto's IFCCA Competition for the Design of Cities project in New York (Benjamin and Libeskind, 1998)—all show radical departures from conventional forms enabled by computer aided design software. (Kolarevic, 2000) provides an overview of these developments, some of which are mentioned in (Perrella, 1998; Toy, 1999).

With these novel forms has come a renewed interest in generating form algorithmically, that is, writing computer programs whose execution results in three dimensional geometry, and using this computational medium to

explore architectural form. Generative systems are a well-established theme in computer-aided architectural design, including the approaches of shape grammar, genetic algorithms, and parametric variation. However, algorithmic form generation has largely remained the province of academic investigation, perhaps because of the level of technical expertise it has required.

Visual arts and music have a long history of exploring algorithmic form generation. Computer music, from the early days of Illiac, has been largely engaged in the development and deployment of sound producing algorithms. Similarly algorithmic generation of images has been a central theme in visual art. Although algorithmic form generation has certainly had a presence in computer-aided architectural design, it is fair to say that other interests have taken a more prominent position.

Investigating new forms for buildings, freed from the constraints of conventional building materials and techniques, architects are beginning to explore how algorithms can be used to generate complex three-dimensional curves, arrangements, and folding of space and material. Algorithmic generation can produce significantly different and more complex forms than conventional CAD.

Despite the power of industrial strength 3D modellers such as CATIA and Maya, many things are difficult to accomplish, and some just can't be done. It remains difficult to use most CAD modellers (Revit notwithstanding) to generate three dimensional forms through parameterisations and more sophisticated algorithms. Algorithmic form generation is familiar in the engineering design disciplines: civil and aeronautical engineering and naval architecture, where three dimensional shapes are more strictly dictated by functional requirements expressed as mathematical equations.

If one wants to generate three dimensional forms algorithmically, one must decide between two main alternatives: On the one hand macro facilities and scripting languages within CAD modellers are relatively easy to learn, but they inherently limit the programs one can write (and hence the forms one can generate). On the other hand, full-fledged programming languages such as C and Java are powerful but they require more effort to learn, and generating 3D geometry also requires attention to many language features that have no direct bearing on form.

1.2 Support for algorithmic form generation

Table 1 distinguishes five levels of support for algorithmic form generation provided by CAD modellers. The simplest modellers provide no support whatsoever for algorithmic generation: models must be constructed directly using the geometric primitives and operations provided on the CAD modeller's menus. Most CAD modelling programs offer macro facilities or

Table 1. Five levels of support for algorithimic generation

1	None	the designer may use only the geometric primitives and operations built in to the modeller.
2	Macros	frequently used sequences of operations can be recorded and replayed, in some cases allowing parameters to be supplied at replay time.
3	Scripting languages	more control over the modeller than recorded macros but which fall short of a full-fledged programming language.
4	Embedded programming language	e.g., AutoCAD's AutoLisp or ArchiCAD's GDL. Access to the modeller's library via a language within the modeller.
5	External programming language	programs typically written in C or Java communicate with and control the modeller. Bentley Microsystem/J

scripting languages. Although a macro facility serves simple tasks well (such as repetitive window patterns or stairs), it is difficult to program more complex operations using only macros. Scripting languages, which have gained wide acceptance in other domains (witness JavaScript and Flash), provide considerably more power than macros but coding more sophisticated tasks becomes quite complex, requiring a specialist programmer. An embedded programming language, like a scripting language, enables the programmer to control and command the modeller from an environment within the CAD program, and allows more powerful constructs than the typical scripting language. Many CAD programs now include an embedded language, and advanced users of these CAD programs enthusiastically endorsetheir modeler's scripting or embedded language. AutoLisp is arguably the best known example. Although the underlying Lisp language is extremely elegant and powerful, Autodesk's implementation was a weak one and the programming environment for developing AutoLisp routines is woefully inadequate by modern standards. Another example of an embedded programming language is ArchiCAD's GDL, which provides access to the modeller's functionality through a BASIC-like language. Although it provides this functionality, the choice of a BASIC programming style limits the language and renders it inelegant: Language design makes an enormous difference. GDL does enable the construction of parametric objects. A fully fledged programming language such as C or Java can be used to write complex form-generating algorithms but it requires more expertise than most designers are willing to commit to acquiring. For a

recent example in visual art see (Berzowska, 1998); an architectural example is Terzides's Java based experiments in morphing (Terzidis, 1999).

Some designers have resorted to using software such as Mathematica or MathCAD to generate three-dimensional surfaces. Mathematical software is good at describing three dimensional surfaces by parameterised functions, but provides limited support (at best) for the basic features that make programming a powerful medium of expression: for example, conditional expressions, iteration, data structures such as sequences and strings. What designers need is a simple way to explore and generate forms algorithmically, without the complexities of a professional programming language.

1.3 FormWriter

FormWriter is an easy-to-use programming language designed especially to allow architects and architecture students to explore algorithmic form generation. The idea of FormWriter is to eliminate complexity without sacrificing the power of programming. FormWriter offers a simple syntax, a unified development environment, and easy access to three-dimensional libraries.

The FormWriter program described below follows in a tradition of novice and domain-oriented programming languages (du Boulay, 1981; Bentley, 1986; Tweed, 1986). The time may now be ripe for a new, architectural, generation of these 'little languages'. Powerful 3D graphics are now available on the desktop and architects and students are better prepared to write code. Like John Maeda's "Design By Numbers" little language for graphic desingers, (Maeda, 1999) the design of FormWriter derives from experience with an earlier generation of novice programming languages at the MIT Logo Project (Papert, 1980).

With only a few lines of code a designer can generate three dimensional graphics immediately, and within minutes can explore parameterised and conditional construction to generate complex combinations of forms. The graphics environment is integrated with the code editor and programming environment, allowing a designer to explore forms fluidly without the distractions of a code-compile-load-execute cycle.

FormWriter is a domain-specific language for novice programmers; nevertheless it a full-fledged language with constructs for passing arguments and returning values, conditional execution, iteration and recursion.

2. FORMWRITER

Figure 1 shows the FormWriter working environment: at left a window into a 3D space with browsing controls; at right an editor window for writing

code. FormWriter also shows the list of user defined procedures as well as the system built-in primitives.

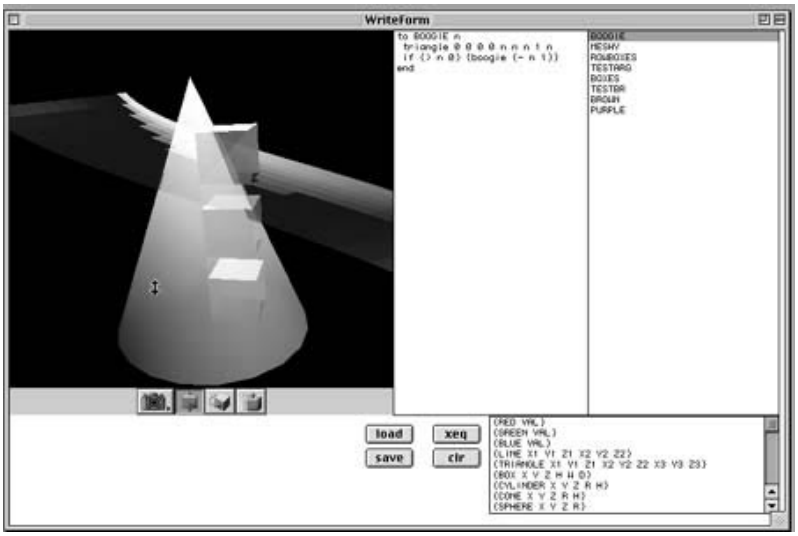

Figure 1. The FormWriter Design Environment

The designer types code in the editor window; to see the result the designer presses the "execute" button. The resulting geometry can be saved as a file and imported into a CAD modeller for further processing. Procedures written in the FormWriter environment can also be saved and reloaded in a future session.

2.1 Simple FormWriter commands

The first thing to do with FormWriter is to generate 3D forms by writing code directly, without defining any procedures. Figure 2 shows the result of a few minutes of play with FormWriter along with the lines of code that generated it. FormWriter's primitive procedures to generate geometry (triangle, cone, box, sphere, cylinder) take dimensions as parameters; the forms are positioned by moving the 3D "flying turtle" forward between the primitive geometry generating procedure calls.

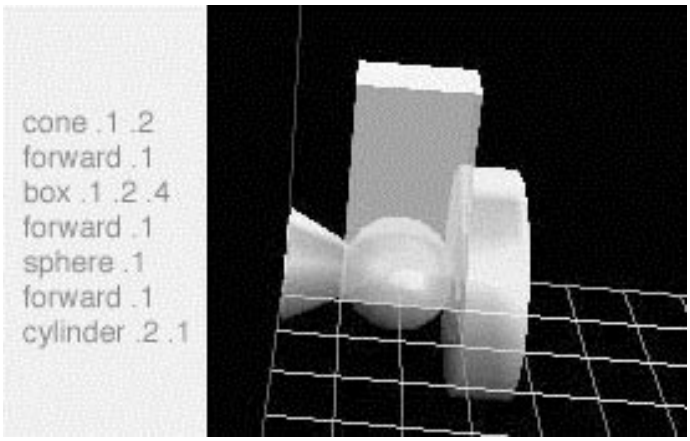

Figure 2. Generating simple forms directly.

Figure 3 shows how the 3D (flying) turtle is used to position and orient five cylinders. The flying turtle can move forward and back, and turn (right and left), pitch (up and down), and roll (side to side). FormWriter inserts each geometric primitive at the current position and orientation of the flying turtle, so as the turtle moves forward (.2 units) and pitches up (30 degrees), between calls to the "cylinder" primitive procedure, each cylinder is translated and rotated in space.

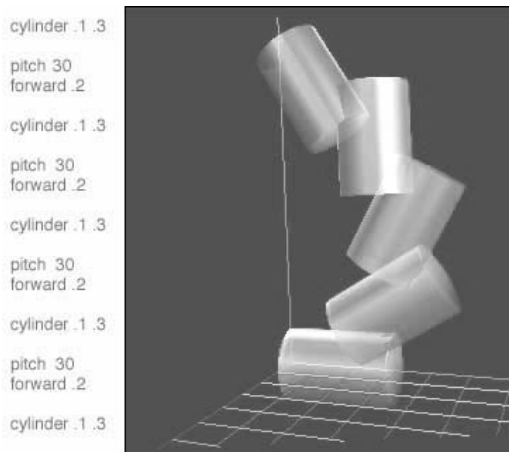

Figure 3. Positioning cylinders in space using the 3D turtle (pitch and forward).

2.2 A first FormWriter procedure

After exploring some simple primitives, and becoming familiar with the flying turtle as a means to position forms in space, the next thing to do is write FormWriter procedures. FormWriter employs the same editor wiindow for defining procedures as for executing commands directly. Each of the forms in Figure 4(a-d) were produced by iterativse calls to the user-defined **one_box** procedure, executing the following line:

repeat (i,10) [one_box()]

Each of the forms used a slightly different definition for the **one_box** procedure as shown in Table 2 below - adding first a turn, then a roll, and finally a pitch instruction to the flying turtle to produce the row of boxes (4a), the turning row(4b), the twisting turning row(4c), and the helix in figure 4(d). The **repeat** statement iterates over a program statement (the call to **one_box**) binding a variable (in this case **i**) to the iteration count.

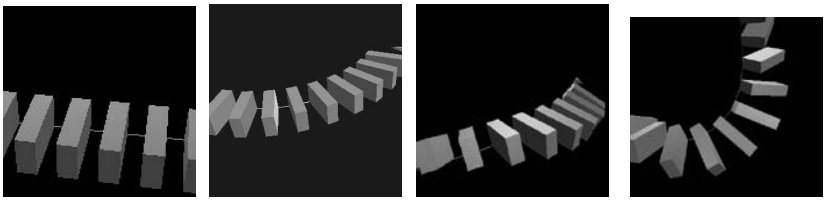

Figure 4. (a) A row of boxes; (b) turning boxes; (c) twisting boxes; (d) boxes helix

Table 2. **one_box** code for the forms in Figure 4

to one_box() box(.4,.1,.2) forward (.2) end	to one_box() box(.4,.1,.2) forward (.2) right(10) end	to one_box() box(.4,.1,.2) forward (.2) right(10) roll(10) end	to one_box() box(.4,.1,.2) forward (.2) right(20) roll(20) pitch(20) end

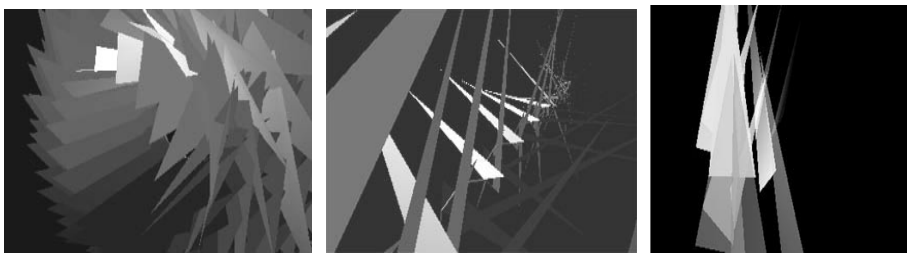

Figure 5. Triangle variations: Orchid, Spiny Tree, and Cathedral

Figure 5 shows some variations produced by a similar program, this time drawing a triangle and written with an input parameter - the code appears in Table 3. TRI is a simple procedure that sets the display color (red, green, and blue values scaled from 0-1) and traces a triangle polygon from the current position and orientation of the flying turtle. The dimensions of the triangle depend on the parameter N that is passed to the TRI procedure. TRIMANY is a recursive procedures that takes one parameter N, calls TRI with that value (resulting in varying size triangles), and then moves forward a distance depending on N, then rolls, pitches, before calling itself recursively, decrementing N until N goes to zero. Slight variations on this code produced the illustrations in Figure 5.

Table 3. Recursive parametric code for generating triangle forms

```
TO TRIMANY (N)
 TRI(N)
 FORWARD((N/ 20)  )
 ROLL(20)
 PITCH(10)
 if (N> 0)   then TRIMANY((N- 1)  )
end
```

```
TO TRI (N)
COLOR (.6,.1,.7)
 TRIANGLE(0, 0, N, N, 0, 0)
 COLOR(.1,.7,.3)
 TRIANGLE(0, N, N, N, 0, 0)
end
```

3. LANGUAGE DESIGN

FormWriter avoids the complexities of languages such as C and Java while providing the full computational power of a functional programming language. Three key features of FormWriter make it powerful and easy to use. These are:

1. immediate results, reducing the four-step edit, compile, load, and execute cycle to a 2-step edit and run cycle;
2. variables are not typed; that is, programmers need not declare variables as integer, string, etc.

3. three dimensional space is described differentially, hiding from the programmer the complexities of transformation matrices and coordinate systems.

The language provides a full complement of control structures: iteration (repeat) and conditional execution (if-then-else) as well as recursion; and data types, including numbers, sysmbols, sequences (lists).

3.1 Immediate results

FormWriter provides immediate results: The programmer writes and runs the code in the same environment. Most compiled programming languages such as C and Java require a four step cycle: (1) edit source code; (2) compile source code; (3) load compiled (object) code; (4) execute the program. (Many integrated development environments enable these four steps to take place in a single environment, but these environments are typically quite complex, providing more advanced programming and debugging tools than a novice is prepared to deal with..) One of the earliest novice programming languages, BASIC (Beginner's All-purpose Symbolic Instruction Code), replaced this four step process with a two step integrated edit and execute cycle. BASIC ran as an interpreted language and programmers could write, run, and debug code in a single environment. When the BASIC interpreter detected an error, or when the program did not behave as expected, the programmer could simply retype the offending lines and RUN the program again.

3.2 Untyped variables

The second key feature of FormWriter is that variables are not typed and need not be declared. In C and Java, every variable must be declared as belonging to a specific type (integer, string, array) before it is used in a program. Some other powerful programming languages—Lisp, for example—do not impose type restrictions. Strongly typed languages are generally thought to help programmers write correct code, because this discipline reveals errors at compile time that might otherwise result in obscure run-time behaviour. For a novice programmer, however, the discipline of declaring variable types before using them imposes an additional distance from writing and debugging code.

3.3 Differential (turtle) geometry

The third key feature is that FormWriter programmers use differential geometry—the flying turtle—to perform translations and rotations of scene geometries. This simplifies considerably the deployment of three-

dimensional forms, because the programmer need not keep track of a coordinate system or understand the transformation matrices that most 3D programming libraries offer. Differential geometry for computer graphics was introduced in the novice programming language Logo: It is one of that language's "powerful ideas" and opens the door to interesting experiments in mathematics and graphics (Abelson and diSessa, 1981). The following program to draw a circle reveals the power and simplicity of this approach:

```
to circle
   repeat [forward 1  r ight 1]
end
```

The turtle traces out a circle by repeatedly moving one step forward and turning one degree to the right. By adopting a local frame of reference the programmer needs no recourse to coordinates, trigonometry, or the (cartestian or polar) equation of a circle: only a common-sense experiential knowledge of space.

3.4 Syntax

The careful reader will have noticed some variation in the syntax used in the examples in this paper. We are exploring three alternative syntaxes for FormWriter: a C-like syntax with infix operators, a Lisp syntax, and a Logo-like syntax. Each has its advantages and disadvantages. We currently support all three syntax variations. User programs are first translated into Lisp and then executed by the Macintosh Common Lisp environment.

The C-like syntax will be familiar to anyone who has had contact with a conventional programming language like C or Java. The Lisp syntax has the simplest syntax rules. The Logo syntax is easiest to write for simple programs, but imposes some constraints on functional composition. Table 4 below shows equivalent statements in the three syntax variants.

Table 4. Syntax alternatives for FormWriter

Logo-style	C-style	Lisp-style
red 1	red (1)	(red 1)
to spiral :dist :angle :geom pitch :angle roll :angle right :angle forward :dist eval :geometry spiral :dist :angle :geom end	to spiral(dist, angle, geom) pitch(angle) roll(angle) right(angle) forward(dist) eval (geometry) spiral(dist, angle,geom) end	(to spiral (dist, angle, geom) (pitch angle) (roll angle) (right angle) (forward dist) (eval geometry) (spiral dist angle geom) end)

4. DISCUSSION

A debate has endured for many years as to whether schools teaching computer aided design should teach computer programming. Opponents of teaching programming argue that architects need to learn to use CAD tools effectively in design, and that building software is best left to professional programmers. Proponents of teaching programming argue that it is essential to understand how computers do what they do and to 'open the black box' of computation even for people who will not become professional programmers.

Perhaps the debate is academic. Recently, a surprising number of architecture students—novice programmers *par excellence*— have embraced the coding details of HTML, JavaScript, Lingo, Flash. They want to produce the effects that they can obtain with these languages and to produce these effects they are willing to put up with an amazing amount of complexity.

4.1 Implementation status

The first version of FormWriter is written in Macintosh Common Lisp and uses the QuickDraw3D API to display three-dimensional geometry. The current development version is replacing QuickDraw3D by OpenGL. We are prototyping version of FormWriter written in Java that runs in a browser, producing VRML code that is displayed using a VRML plug-in or Java3D.

5. ACKNOWLEDGEMENTS

The current version FormWriter is written in Macintosh Common Lisp (MCL) and employs "user contributed code" written by Mark Kantrowitz (infix parsing), and John Wiseman (QuickDraw3D). The OpenGL version currently under development employs the OpenGL API written by Alex Repenning of Agentsheets Inc. The Java applet version is being written by Thomas Jung.

6. REFERENCES

Abelson, H. and A. diSessa, 1981, *Turtle Geometry*, MIT Press, Cambridge, MA.

Benjamin, A. and D. Libeskind, 1998, *Reiser + Umemoto : Recent Projects*, John Wiley & Son Ltd, London.

Bentley, J. L., 1986, " Little Languages -- Programming Pearls." *Communications of the ACM.* **29**(August), p. 711-721.

Berzowska, J., 1998, *Algorithmic Expressionism.* Media Laboratory, Cambridge, MA, MIT.

du Boulay, J. B. H., O'Shea, T., and Monk, J., 1981, “The black box inside the glass box. Presenting computing concepts to novices.” *International Journal on Man-Machine Studies* **14**(3), p. 237-249.

Kolarevic, B., 2000, "Digital Architectures", In *Proc. ACADIA 2000*, ACADIA, Washington, DC (forthcoming).

Lynn, G., 1998, *Animate Form*, Princeton University Press, Princeton, NJ.

Maeda, J., 1999, *Design By Numbers*, MIT Press, Cambridge, MA.

Papert, S., 1980, *Mindstorms, children, computers and powerful ideas*, Basic Books, New York.

Perrella, S., ed. 1998, *Hypersurface Architecture (Architectural Design Profiles,133)*, John Wiley & Son Ltd;, London.

Terzidis, K., 1999, "Experiments on Morphing Systems", In *III Congreso Iberoamericano de Grafico Digital [SIGRADI Conference Proceedings] Montevideo (Uruguay)*, SIGRADI, p. 149-151.

Toy, M., ed. 1999, *Architects in Cyberspace II (Architectural Design Profile, No 136)*, John Wiley & Son Ltd, London.

Tweed, C., 1986, "A Computing Environment for CAAD Education", In *Teaching and Research Experience with CAAD [4th eCAADe Conference Proceedings]*, eCAADe, p. 136-145.