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.
Introduction

There has recently been a resurgence of interest in new methods of generating form. Stimulated, in part, by work such as Frank Gehry’s Guggenheim museum in Bilbao and the Experimental Music Project in Seattle, and Greg Lynn’s ‘blob’ architecture, many architects are seeking new methods to stimulate the creative generation of form. Among the methods architects are exploring is using algorithms to generate complex three-dimensional curves, arrangements, and foldings of space and material. In the case of Gehry, these forms result are made using software designed for aeronautical design; in the case of Lynn, they are generated by programmers who work with the architect to devise form-generating algorithms. In both instances, conventional architectural CAD modeling tools would not serve. Algorithmic generation can produce significantly different and more complex forms than conventional CAD, and emerging application of computer assisted manufacturing (CAM) techniques in architectural construction now make it possible to generate buildings whose form could not be easily described in any other way.

Producing three-dimensional computer graphics models has been made easy by the development of powerful modeling software that allows architects to shape and sculpt three dimensional material and space using ‘direct manipulation’ operations. However, it has been difficult to control the generation of three-dimensional forms through complex parameterizations and more sophisticated algorithms. This capability has been more common in civil, aeronautical, and naval architecture. Although most architectural modeling programs offer macro facilities or embedded scripting languages, these are not especially easy to use, and in some cases, not well integrated into the modeling environment. On the other hand, although full-fledged programming languages such as C and Java can be used to write complex form-generating algorithms, they also require significant amounts of knowledge and effort in order to generate even simple forms. Some designers have resorted to using software such as Mathematica to generate three-dimensional forms, but these packages are oriented for other purposes than design, and are thus less than optimal. What is needed is a simple way for designers to explore and generate forms algorithmically, without having to learn a lot of needless programming complexity.
For these reasons we have been developing FormWriter, an easy-to-use programming language designed especially to allow architects and architecture students to explore generating forms algorithmically. A designer can generate three dimensional graphics immediately with only a few lines of code, and within minutes can explore parameterized and conditional construction of forms to generate complex combinations of shapes. The graphics environment is integrated with the code editor and programming environment which allows a designer to explore forms fluidly without an annoying code-compile-view cycle. This book describes the FormWriter language and our experience with using it as a medium for generating form.

FormWriter is a language for novice programmers, designed as “low threshold; no ceiling.” Although its ease of use makes it simple for non-programmers to use, it is nevertheless a powerful language, with constructs for passing arguments and returning values, conditional execution, iteration and recursion.
FormWriter Environment

The first step to get acquainted with FormWriter is to see the software environment we will use in this book. Figure 1.1 shows the FormWriter working environment: on the left is a display window with browsing controls, where the visual output will be shown in 3-D; on the right is an editor window for writing code. FormWriter also shows the list of user defined procedures as well as the system built-in primitives. 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 for further processing. FormWriter also allows the designer to load saved files for further modification. When the designer opens a saved file, the code of that file will be shown on the editor window.
The Flying Turtle

As seen on the display window after the designer opens FormWriter, a fixed global coordinate that indicates + X, + Y, and + Z axes appears on the display window Figure 1.2. The designer can also choose not to show the coordinate. The initial view on the display window shows X-Y plane with + Z axis facing to users. Right now users should be able to see the flying turtle, actually a triangle frame, standing at the origin with the head pointing to the + Y direction and waiting for our commands to make something happen. The flying turtle can move forward and back, and turn (right and left), pitch (up and down), and roll (side to side).
Simple FormWriter Commands

The first thing to do with FormWriter is to generate 3-D forms by writing code directly, without defining any procedures. As you might be curious: can we draw something by defining procedures? The answer is Yes. However, we will start with the most fundamental operations first, then move to more advanced things later. We will get back to ideas about defining procedures later in this Chapter.
Basic Turtle Operation

The easiest way to generate graphics in FormWriter environment is to directly drive 3-D (flying) turtle moving. The turtle can move around the 3-D world. When it moves from one point to the next, it draws a trail between those two points. However, the turtle trail is not always shown. The trail will be always drawn with the system default condition. However, we can command FormWriter not to show the turtle trail. We will address this issue later in the book.

There are three different types of turtle trails that designers can choose. The system default trail type is the Line turtle trail. Here we first introduce how to create graphics only by using the Line turtle trail. When you drive the turtle moving forwards, FormWriter draws a line that connects the previous turtle position to the current turtle position, as shown in Figure 1.3.

FormWriter Syntax System

Before we move on to other examples and introduce other FormWriter commands, it is necessary to make everyone clear about how to write the code.

FormWriter currently uses a Lisp syntax system that has a set of simple rules. The first rule is that each single command is represented by an expression that is always enclosed in a pair of parentheses. In the previous example, the expression \( \text{forward 2} \), which actually commands FormWriter to draw a line, starts with an open-parenthesis and ends with a close-parenthesis. Moreover, in the expression \( \text{forward 2} \), the \text{forward} is actually an operator, and the number 2 is a parameter that gives a value to the operator.

Here is another example of the FormWriter syntax system. When we do a simple calculation: 2 plus 3, we would naturally write the expression as \( 2 + 3 \). But in FormWriter we should write it as \( (+ 2 3) \), which starts with an open-parenthesis, followed by the operator +, and then two parameters 2 and 3, and finally with a close-parenthesis. The careful reader would have noticed that the operator \text{forward} takes only one parameter, but the operator + takes two. Yes! Some operators do need not just one parameter but many to complete the operation. We will get back to this issue later in this Chapter.

An expression actually can contain other expressions. For example, the expression \( \text{forward (+ 1 1)} \), which contains another expression \( (+ 1 1) \), does exactly the same thing as the expression \( \text{forward 2} \) does.
More Examples

By far, you should have got familiar a little more with the FormWriter syntax system. So let’s take a look at more examples about basic turtle operations. Besides moving forwards, the turtle can also move backwards, Figure 1.4. You can also drive the turtle turning right or left. Figure 1.5 shows the result of combining commands forward and right. In addition, as mentioned before, the turtle can move around the 3D world. There are other two commands, pitch and roll, which enable designers to really master the turtle to fly 3-dimensionally. Figure 1.6 shows that the turtle start to move 3-dimensionally by using commands pitch and roll.
Let's start to really draw some graphics using only the Line turtle trail. We can drive the turtle to enclose a loop in order to form a shape. In Figure 1.7, a square frame is drawn by moving the turtle forwards by 2 and then turning right by 90 degrees four times respectively.

We can draw something more complicated here. How about SpaceNeedle and a little skyline of the downtown Seattle, Figure 1.8.
Draw in 3-D by the Turtle Trail

As we mentioned before, there are three different types of turtle trail that designers can choose. In all of the previous examples, we only used the Line turtle trail to generate graphics. In this section, we will introduce another two types of turtle trail, the Box turtle trail and the Cylinder turtle trail.

First we introduce a new command that sets up the turtle trail that will be shown on the display window.

(set-turtle-trail trail_type)
trail_type: line, box, cylinder

Again, the entire command is enclosed in a pair of parentheses. The operator set-turtle-trail tells FormWriter that “I am going to set up a certain type of turtle trail”. But what kind of type is FormWriter supposed to set up? Therefore, we have to give a parameter trail_type to the operator, then FormWriter knows what to do. Let’s start to play with turtle trail. Say we want to set up the turtle trail to the Box.

(set-turtle-trail box)

So when we drive turtle moving from one point to the next point, FormWriter will generate a box between those two points. In Figure 1.9, two partially overlapped boxes are drawn by moving turtle forwards by 2, then turning right by 90, and finally moving forwards again by 2.
You might be curious about what the dimension of the trail box is? FormWriter sets the default dimension of the trail box as width 2 by height 2. However, FormWriter allows us to define the dimension of the trail box. The command \texttt{set-trail-box} does this task.

\texttt{(set-trail-box width height)}

Figure 1.10 shows how the command \texttt{set-trail-box} works. We can also set up the turtle trail to the Cylinder.

\texttt{(set-turtle-trail cylinder)}

\texttt{(set-trail-cylinder)}

Figure 1.11 shows FormWriter generates three cylinders by moving and turning the flying turtle. Similarly, there is a default value of the dimension of the trail cylinder. The default value of the radius of the trail cylinder is 1. Again, we can define the dimension of the trail cylinder by ourselves.

\texttt{(set-trail-cylinder radius)}

Let’s change the dimension of the trail cylinder and draw some different cylinders with different sizes, Figure 1.12.
Pen Control

By far, we understand how to generate graphics or 3-D models by driving the turtle moving around the 3-D world. However, you might have thought that sometimes when we move the turtle, we do not really want to draw anything but just drive the turtle “jumping” to another certain point and then draw something. How can we possibly do that? Here we introduce the idea of the pen control. There are two pen-control commands:

\[(\text{penup})\]
\[(\text{pendown})\]

The idea is simple. As the word “pen-up” implies, when the command \(\text{penup}\) is called, FormWriter will not draw the turtle trail while the turtle moves. On the contrary, the command \(\text{pendown}\) will call FormWriter to always draw the turtle trail. The initial setting in FormWriter is \(\text{pendown}\). In Figure 1.13, we first move the turtle forwards by 2 with the Box turtle trail being set, and then we call the command \(\text{penup}\) and move the turtle forwards by another 2, FormWriter will only move the turtle without drawing the trail. Finally we move the turtle again to get another box. Figure 1.14 shows a red-cross drawn by playing the two commands \(\text{penup}\) and \(\text{pendown}\).
3-D Geometric Primitives

In addition to drawing 3-D geometry by the turtle trail, FormWriter actually provides two other ways for the designers to generate 3-D objects. One way is to use 3-D geometric primitives; the other is to draw 2-D polygons. There are four different types of 3-D geometric primitives: Box, Cylinder, Cone, and Sphere.

We can think of using 3-D geometric primitives in this way: we first drive the turtle moving to a certain position, and then we “drop” a 3-D object (a primitive) at this position. Figure 1.15, 16, 17, and 18 respectively shows that we individually drop a box, a cylinder, a cone, and a sphere at the origin (0,0,0). Let’s play with those 3-D geometric primitives and draw something fun. Figure 1.19 shows how easy it is to draw a robot by combining geometric primitives.
Colors

To those who want to draw really cool graphics, being able to apply different colors to the graphics should be essential. Therefore, here we introduce another command set-color that allows us to specify a color, or say a color scheme.

(set-color red green blue)
red: 0~100
green: 0~100
blue: 0~100

Before we start to play with colors, we should be aware of one thing: whenever a particular geometric object is drawn, it is drawn using the currently specified color. In general, we can first set the color or coloring scheme and then draw the objects. Until the color is changed, all objects are drawn in that color. The initial system default color is red \((100, 0, 0)\). Therefore everything will be drawn in red color unless we change it to a different color.

To set a color, use the command set-color. It takes three parameters, all of which are numbers between 0.0 and 100.0. The parameters are, in order, the red, green, and blue “components” of the color. You can think of these three values as specifying a “mix” of colors: 0.0 means don’t use any of that component, on the other hand, 100.0 means use all you can of that component. Here are eight basic colors:

(set-color 100 0 0); red
(set-color 0 100 0); green
(set-color 0 0 100); blue
(set-color 100 100 0); yellow
(set-color 100 0 100); magenta
(set-color 0 100 100); cyan
(set-color 100 100 100); white
(set-color 0 0 0); black

Ok, let’s put some different color on the robot that we created just before. Figure 1.20 shows a colorful robot with several different coloring schemes.

(set-color 100 0 0)
(penup)
(box 4 2 4)
(set-color 0 100 0)
(right 90)
(forward 1.3)
(right 90)
(cylinder 0.5 5)
(home)
(left 90)
(forward 1.3)
(left 90)
(cylinder 0.5 5)
(set-color 0 0 100)
(home)
(forward 4)
(cylinder 0.5 0.5)
(forward 2)
(sphere 1.5)
(forward 1)
(set-color 0 100 100)
(cone 1.5 2)
(back 1)
(pitch 90)
(forward 1.5)
(cone 0.4 0.4)
(set-color 100 100 0)
(home)
(forward 3.5)
(right 90)
(forward 2)
(cylinder 0.4 3)
(home)
(forward 3.5)
(left 90)
(forward 2)
(right 5)
(cylinder 0.4 3)
Polygons

Polygons are the areas enclosed by single closed loops of line segments. In general, polygons can be complicated, so FormWriter makes some restrictions on what constitutes a primitive polygon. First, the edges of polygons cannot intersect. Second, polygons must be convex. However, FormWriter does not restrict the number of line segments making up the boundary of a convex polygon.

A triangle, which consists of three line segments, is the simplest polygon. FormWriter provides users a command triangle to simply draw a triangle polygon.

(triangle x1 y1 z1 x2 y2 z2)

The command triangle takes six parameters, which represent two of three points constituting a triangle. The third point is the current origin where the turtle is standing on, Figure 1.21.
As mentioned before, FormWriter does not restrict the number of line segments making up a convex polygon. However, how is it possible to draw a polygon by a number of line segments. We can simply drive the turtle moving forwards and turning by certain degrees in order to “enclose” a loop to form a polygon, just as the square example we showed before. However, in the example, we actually drew a square frame; not a polygon. We have to do something else other than just driving the turtle. FormWriter provides users a pair of commands, which should always come together, to draw a polygon.

\[(\text{start-poly})\]
\[(\text{end-poly})\]

To draw a polygon, we first call the command \(\text{start-poly}\), and then start to drive the turtle. Finally we call the command \(\text{end-poly}\) to make FormWriter render the polygon for us. Figure 1.22 shows that a square polygon is drawn by calling the two commands and driving the turtle following the same procedure as in square frame example. In Figure 1.23, a polygon in an irregular shape and with several line segments is simply drawn by driving the turtle moving. One thing the users should be notified: we do not have to always drive the turtle back to the starting point to enclose a polygon. Instead we can stop at the last point that encloses a convex loop. Let’s play with the two commands, Figure 1.24.
Repeat

In several previous examples, we have drawn a square frame (Figure 1.7), a square polygon (Figure 1.22), or several diamonds (Figure 1.24) by driving the turtle moving forwards and turning repeatedly several times in order to form a loop. As you would argue, why not we call a command to repeat some certain processes by some certain times. In fact, FormWriter does provide a command \texttt{repeat} allowing the users to repeat drawing processes.

\begin{verbatim}
(repeat indicator count
  (action 1)
  (action 2)
  ...
  (action n))
\end{verbatim}

We can now re-write the code using the command \texttt{repeat} to draw the same graphics as in previous examples, Figure 1.25, Figure 1.26.

\begin{verbatim}
(repeat i 4
  (forward 2)
  (right 90))
\end{verbatim}

\begin{verbatim}
(start-poly)
(repeat i 4
  (forward 2)
  (right 90))
(end-poly)
\end{verbatim}
Procedure

By far, we have explored some simple FormWriter commands, including basic turtle operations, turtle trail operations, 3-D geometric primitives, and polygons. All examples by far were drawn by writing code directly without defining any repeating procedure. In other words, we call each single command and “command” FormWriter to do the task correspondingly. For example, in Figure 1.24, we have drawn four identical polygons by calling start-poly and end-poly four times. As you would argue, why not we define a procedure to repeat some certain processes. In fact, FormWriter does allow designers to define procedures that will be used repeatedly. The command define does this task.

```
(define name (parameters)
(action 1)
(action 2)
...(action n))
```

Let’s re-draw the graphic in Figure 1.24 by defining a procedure, Figure 1.27. A designer-defined procedure, named diamond, actually draws the diamond polygon. Instead of repeating the entire processes, the designers only call the defined procedure to draw the repeating polygon.

```
(define diamond ()
  (start-poly)
  (left 30)
  (forward 1)
  (right 60)
  (forward 1)
  (right 120)
  (forward 1)
  (end-poly))

(sphere 1)
(forward 1)
(diamond)
(home)
(right 90)
(forward 1)
(diamond)
(home)
(left 90)
(forward 1)
(diamond)
(home)
(right 180)
(forward 1)
(diamond)
(home)
```

Figure 1.27
A procedure being defined before can be called by other procedures. Let’s define a procedure, called \texttt{square}, which draws a square frame Figure 1.28; then we define a new procedure, called \texttt{cube}, which calls the procedure \texttt{square} to draw a cube frame Figure 1.29. Given the ability to define and recall procedures, we are able to avoid the repeated coding and have FormWriter generate geometry more efficiently.

\begin{verbatim}
(define square ()
  (repeat i 4
    (forward 2)
    (right 90)))
(square)

(define cube ()
  (repeat i 4
    (square)
    (forward 2)
    (pitch 90)))
(cube)
\end{verbatim}
Variables

We now know how to define a procedure to repeatedly generate graphics. However, sometimes we would like to generate graphics with the same shapes or composition but in different size or dimension. Of course, we can define another procedure. But we should be able to find out an easier way. Here we introduce the idea of variables.

In the previous example, to draw a diamond, we drive the turtle moving forwards by 2 four times. The distance 2 is actually the dimension of four sides of the diamond. If we want to generate a diamond with distance 3 on all sides, we have to change the value that we drive the turtle moving forwards. So, let’s modify the original procedure a little. Now we redefine a new procedure that takes a argument called size. Every time we call the new procedure to make a diamond, we have to give the procedure a value of the dimension, Figure 1.30.

```
(define diamond-size (size)
  (start-poly)
  (left 30)  (forward size)  (right 60)  (forward size)
  (right 120)  (forward size)  (end-poly))
```

```
(sphere 1)  (forward 1)  (diamond-size 2)  (home)
(right 90)  (forward 1)  (diamond-size 2)  (home)
(left 90)  (forward 1)  (diamond-size 2)  (home)
(right 180)  (forward 1)  (diamond-size 2)  (home)
```

Figure 1.30
We are allowed to use as many arguments as we need to define a procedure. In Figure 1.31, the procedure polygon takes two arguments, side and size, and allows us to make different polygons in different shape and dimension. Let's play with multi-variables again. In Figure 1.32, we first define a procedure moveturn that carries two arguments, then we define a new procedure circle, which also carries two arguments, to call the predefined procedure moveturn to make a circle. We can see that the two arguments in the procedure circle are actually passed to the procedure moveturn to make move and turn step by step.

(define polygon (side size)
  (repeat i side
    (forward size)
    (right (/ 360 side))))

(polygon 6 2)

(define moveturn (step turn)
  (forward step)
  (right turn))

(define circle (step turn)
  (repeat i 50
    (moveturn step turn)))

(circle 0.1 5)