

Life Goes On

by Robert M. Ryan

The art and science of programming a computer involves much more than simply stringing language statements together. The most important steps in programming come long before you enter a PRINT or IF/THEN statement.

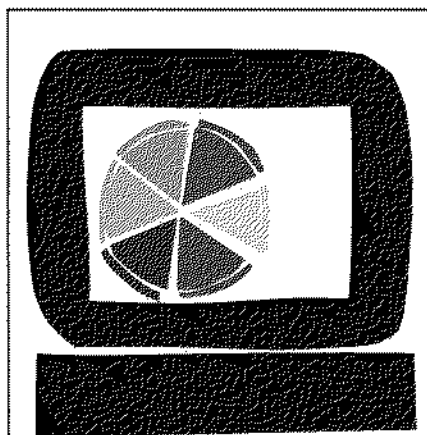
Before you code a program, you've got to define what you want a program to do, the output the program should produce, and the input necessary to produce that output. Then you have to come up with an algorithm that'll convert the input into the appropriate output. In this month's column, I'll use the electronic game of Life to demonstrate the steps needed to complete a programming project successfully. (Follow along in the accompanying AC/BASIC Program listing.)

DEFINING THE PROBLEM

The game of Life is one of the oldest and most fascinating computer games. It was created by John Conway, a British mathematics professor, and popularized in the early 1970s by Martin Gardner in his "Mathematical Games" column in *Scientific American*. Its enduring popularity is a reflection of the complexity and diversity a very simple set of rules can generate.

The game of Life simulates a simple biological system consisting of a grid of cells—you can think of each cell as a biological niche. Each cell therefore has eight neighboring cells; one above, one below, one on either side, and four on the diagonals. At any time, a cell can either contain life or be devoid of life. (It can be either on or off.) Whether a cell is on or off is completely dependent on the following rules of the Life system:

- 1) If, in the previous generation, the total number of neighboring cells that were on was zero or one, then the cell will be off.
- 2) If the total number of neighbors with life was two, then the cell maintains the



Simulating a simple biological system in the game of Life will show you how to define a programming problem and outline the solution.

John Muller

condition, either on or off, that it had in the previous generation.

- 3) If the total number of neighbors with life was three, then the cell has life in the current generation.

- 4) If the total number of neighbors with life was four or more, then the cell is off for the current generation.

In the Life system, time is discrete; you measure it in generations. You provide the first generation, called the *parent* or *seed* generation. The rules of the Life system produce subsequent generations; the game's challenge is to create dynamic yet stable systems that last for hundreds of generations. Your challenge is to create a program that adheres to the rules of Life.

SPECIFYING OUTPUT

Since the game consists of a rectangular

grid of on and off cells, you can easily represent the output of the program with a window on the IIGS graphics screen. To keep generation-processing time short, I use a window that's smaller than the screen. Whenever a cell is off, I leave it blank on the output display. When a cell is on, I represent it with an asterisk.

The big question, of course, is how to generate output. You achieve this by examining the Life universe and printing asterisks wherever you find life. For this you need an internal representation of the Life universe; you'll find it in two arrays, *parent%* and *filial%*, both two-dimensional.

Parent% contains the current condition of the Life system. To create the next generation, your program examines each cell in *parent%* according to the rules of Life, then stores the results in *filial%* and displays them in the output window. The program then copies *filial%* into *parent%* and repeats the process.

The exact representation of the Life universe in an array is quite simple: Each element of *parent%* and *filial%* corresponds to a cell in that universe. If an array element such as (5,13) contains a 0 (zero), then that cell doesn't contain life, and the corresponding location in the output window will be blank. If the element contains a 1 (one), then the cell has life, and the corresponding location on the output screen contains an asterisk.

To generate the output of the Life program, you simply test the elements of the output array (*filial%*) and print a blank where the array value is 0, and an asterisk where the value is 1.

PROGRAM INPUT

Because both arrays are initialized to 0, you need some way to seed the system with life. I use a very primitive seeding routine

in my program. I could have trapped menu and mouse events, but they slow down program execution, and I wanted to produce generations as fast as possible.

To speed up the process, my program lets you seed 12 cells with life by simply clicking on them with the mouse. The program polls the mouse and puts a 1 into the parent% array element that corresponds to the output location you clicked. You can change the number of seeds or modify the seeding routine entirely if you like.

INPUT TO OUTPUT

Once you have the seed, you must process the next generation. I use nested FOR/NEXT loops to add the values of the neighbors of each cell in the parent% array. I then apply the system rules in series of IF/THEN statements to see whether the corresponding cell in filial% is on or off.

"The game of Life is one of the oldest and most fascinating computer games. Its enduring popularity is a reflection of the complexity and diversity a very simple set of rules can generate."

Ideally, the Life system is infinite. Unfortunately, your output window isn't. The easiest way to handle the borders of the display is to keep track of one row or column of cells outside the visible display and to ignore what happens beyond these boundary rows and columns. Thus, although my arrays start with element (0,0), I don't display row 0 or column 0. They're present in the Life universe, so all displayed cells have eight neighbors.

Program listing. The game of Life.

```
rem IIGS Game of Life by Bob Ryan
rem Copyright 1988, inCider
rem A/C BASIC compile with c, m, and u options

dim parent%(61,15), filial%(61,15)
window 1,"life",{44,80}-{464,192}
menu 1,0,1,"project"
menu 1,1,1,"quit"
menu 2,0,0,""
gen% = 1
gosub GETSEED

LOOP:
  gosub SWAPIT
  gosub COMPUTE
  gosub DISPLAY
  gosub SWITCH
  if menu(0) <> 0 then
    locate 1,1
    print "Total number of generations: ";gen%
    delay! = timer
    while timer < delay! + 3
      wend
    stop
  end if
  gen% = gen%+1
  goto LOOP

DISPLAY:
  cls
  for y% = 1 to 14
    for x% = 1 to 60
      if filial%(x%,y%) = 1 then
        locate y%,x%
        print "*";
      end if
    next x%
  next y%
  return

SWITCH:
  for x% = 0 to 61
    for y% = 0 to 15
      parent%(x%,y%) = filial%(x%,y%)
    next y%
  next x%
  return

GETSEED:
  c = 0
MLOOP:
  delay! = timer
  while timer < delay! + 1.25
    wend
  if mouse(0) = 0 then MLOOP
  col% = int(mouse(5)/8+.5)
  row% = int(mouse(6)/8+.5)
  locate row%,col%
  print "*";
  parent%(col%,row%) = 1
  c = c + 1
  if c = 12 then return
  goto MLOOP

SWAPIT:
  for x% = 0 to 61
    parent%(x%,14) = parent%(x%,0)
    parent%(x%,1) = parent%(x%,15)
  next x%
  for y% = 0 to 15
    parent%(1,y%) = parent%(61,y%)
    parent%(60,y%) = parent%(0,y%)
  next y%
  return

COMPUTE:
  for x% = 1 to 60
    for y% = 1 to 14
      sum1% = parent%(x%-1,y%) + parent%(x%,y%-1) + parent%(x%+1,y%)
      sum2% = parent%(x%,y%+1) + parent%(x%-1,y%-1) + parent%(x%+1,y%-1)
      sum3% = parent%(x%-1,y%+1) + parent%(x%+1,y%+1)
      sum% = sum1% + sum2% + sum3%
      if sum% = 2 then
        filial%(x%,y%) = parent%(x%,y%)
      elseif sum% = 3 then
        filial%(x%,y%) = 1
      else filial%(x%,y%) = 0
      end if
    next y%
  next x%
  return
end
```

End

It's a simple solution, but it's not very satisfying. A better method, the one I use in the listing, is to make the Life system *toroidal*—that is, to make it wrap around itself so that the top row of cells borders the bottom row and the left side borders the right. This way, the Life system is completely self-contained.

Achieving a toroidal effect is quite simple—I use a *swap* routine. Here, after the program calculates a generation, displays it, and copies it back to the parent% array, I copy the top visible row of cells to the boundary row at the bottom of the system and the last visible row to the boundary row at the top. I perform a similar swap between the left and right sides. The result, in effect, is that each of the visible cells around the border of the window has eight neighbors, and their invisible neighbors come from the opposite side of the display.

IMPLEMENTATION

Once you know what your output should look like, what the input is, and how you get from one to the other, you can begin to write the program. The accompanying **Program listing** contains the code I wrote to create the Life system.

Enjoy the game of Life, and note carefully how I implemented the steps described above. Because of space constraints, I wasn't able to add a lot of chrome to this program, such as succeeding generations displayed in different colors. If you want to see how I implemented color and event trapping in version 2 of the program, download the file **life.extra** from the *inCider* bulletin-board system (603-924-9801). I'll also post the printed edition on *inCider's* BBS.

Remember, you can't code a program you haven't thoroughly thought through. Save yourself a lot of work coding and debugging by putting your effort into defining your problem. You'll be a faster and better programmer for it. ■

Bob Ryan is a technical editor at AmigaWorld magazine (IDG Communications/Peterborough) and is a contributing editor at inCider. Write to him c/o inCider, 80 Elm Street, Peterborough, NH 03458.