Artificial Life

by John Iovine

Creating artificial life forms is one of the most interesting things you can do with your Amiga computer. Artificial life programs are also called Cellular Automation (CA) programs and Genetic algorithms. Whatever you call them, this form of life is electronic and exists only within the confines of the computer system.

Artificial life programs are not merely toys for the technically inclined. These programs are used to successfully model biological organisms and eco-systems. Cellular automations can mimic evolution, co-evolution, the migration of birds, colonies of ants, social order of bees, and bacteria colonies. Chaos algorithms may also be incorporated into the CA programs to add a degree of randomness that increases the accuracy in modeling weather patterns, population growth, and the spread of infectious disease. Artificial life programs are also being developed to optimize neural networks, artificial intelligence, and parallel processors. Computer experiments are underway generating CA programs that will create and wire neural networks patterns. When these programs become successful they essentially will teach themselves.

Genetic Algorithms

Genetic Algorithms are a class of artificial life that evolves in a Darwinian, survival-of-the-fittest, manner. I specify Darwinian because other researchers are busy using CA programs to test co-evolution theories. Typically Genetic Algorithms programs are designed to internally optimize their program code for specific problem solving.

Danny Hills, the founder of Thinking Machines Corporation in Cambridge, Massachusetts, programmed an interesting GA. In Hills Genetic Algorithm experiment he needed one of his company's Connection Machine computers. This super computer contains 65,536 parallel processors that allowed it to quickly simulate 65,536 independent organisms. The artificial organisms used in the experiment were numerical sorting programs.

To mimic evolution the programs could randomly change (mutate) their code. They could also combine their program codes with other program codes (organisms), in a process similar to artificial sex. All new programs created were tested for efficiency in sorting numbers. Only the fittest or most efficient programs survived. This electronic evolution scheme quickly evolved improvements in program code.

Writing Music

Artificial life programs can also write music. Eric Iverson of New Mexico State University created an artificial life program that creates music. This program is not like other music generation programs. Most computerized music generation programs rely on generating a large amount of random notes that are then filtered though an artificially intelligent (AI)

sub-routine. The AI sub-routine section contains a bunch of "rules" that constitute the programmers best ideas of what note sequence can make music.

Artificial life music generation operates differently. The artificial life program eats music fed to it. We could feed the program any type of music from Madonna to Mozart. The program uses the music structure (notes) to generate new structures of notes.

The process I used in the basic program Listing 2 could easily be modified to generate new music from old music, although the basic algorithm is different from the one used by Mr. Iverson.

Cooperative Co-Evolution

Darwin's theory of evolution specifies survival of the fittest, not the nicest. Two Austrian mathematicians decided to have another look at this premise and have discovered, through the use of CA programs, that cooperative and unselfish behavior may play a role in the story of evolution. Karl Sigmund and Martin Nowak's new cooperative co-evolution theory is supported in nature.

For instance, the vampire bat will share their blood meals with unrelated, less fortunate neighbors. A vampire bat is required to consume 50 to 100 percent of its body weight in blood every night. If it fails to feed for two nights in a row, it will die. However, a bat can gain another 12 hours of life and another chance to feed if it is given a regurgitated blood meal by a roost mate. If the bats didn't practice food sharing, their annual mortality would be 82%. With food sharing the number drops down to 24%. Obviously from an evolutionary standpoint this unselfish behavior benefits the propagation of the species. Bats are not the only species to practice unselfish behavior.

Cellular automation programs vary in levels of sophistication. Obviously Danny Hill's Connection Machine CA program is far too complex for us to program onto our simple sequential computers. However, CA programs generally consist of a few simple rules. When the computer rapidly and consistently repeats the rules through generations, complex patterns can evolve that are not written into the program; they develop by themselves. The program’s striking behavioral patterns provide windows of opportunity to study and to model living systems and artificial intelligence.

To help you acquire a knowledge of cellular automation programs, artificial life and genetic algorithms, I have included a few basic language program listings to get you started. The basic language programs, although slow and simple, will provide insight into these programming beasties. These insights would otherwise remain hidden in more complex, compiled, or commercial programs.

Enter Life

No discussion of artificial life would be complete without mentioning John von Neumann’s seminal work in the late 1940s on cellular automata. Martin Gardner’s "Mathematical Games" column in *Scientific American* brought cellular automation to the masses. The "Game of Life" was originally described in *Scientific American* in the early 1970s by Martin

Gardner. John Horton Conway, a British mathematician then at the University of Cambridge in England, created the game. Professor Conway is currently teaching at Princeton University.

The game is played on a large two-dimensional grid of square cells, similar to a checkerboard. Each cell or block of the checkerboard can contain an organism, identified as a dot. Each cell on the checkerboard is surrounded by eight neighboring cells. Four cells are adjacent orthogonally and four cells adjacent diagonally (Figure 1).

In the game of life, an organism dies or reproduces according to a few simple rules derived by Conway. In deriving rules, Conway sought to find the simplest rules that would create populations with the largest diversity and unpredictability. The rules to play one game of life are as follows:

1. Survival. Every organism with two or three neighboring cells that also contain organisms survives to the next generation.
2. Death. Each organism with four or more neighboring organisms dies of overpopulation. Every organism with just one neighbor or no neighbor dies of isolation.
3. Reproduction. Any empty cell next to exactly three neighbors is a birth cell. An organism is placed on it in the following generation.

To start the game an initial pattern of live cells is placed in the grid. The colony of cells may grow into a large population, fall into a cyclic pattern, or die off. The complexity and diversity of the life generated in the game exceeded what anyone ever thought the underlying simplistic rules would create.

Conway discovered an interesting pattern of cells he termed a glider. The glider mutated through four generations returning to its original form displaced diagonally by one cell space. The Life program uses a glider for a starting pattern. It is difficult to see the glider move because the program execution is slow using BASIC. However, Figures 2 thru 6 show five consecutive generations (screen images) of the glider moving. Notice the initial glider pattern in generation 0 is the same as the glider pattern in generation 4. The only difference is that the glider moved diagonally one position. The motion is readily apparent in compiled or assembly language programs.

Game of Life BASIC Listing Explanation

The BASIC version of the game of Life is slow. It may take a minute or more between screen updates. Each screen update represents a new generation.

The arrays OL (old) and NE (new) are two-dimensional, 24 cells (rows) by 70 cells (columns). This two-dimensional array represents the cells of the computer's text screen display that is used. Each position on the text display is a cell that may or may not contain an organism. The variable GN represents the current "generation" number.

The subroutine "Start:" places binary 1s in the NE data array. The starting pattern is a glider.

In the display routine the program steps through the entire NE data array. Each cell of the NE data array is printed onto the screen. If the cell contains a "0" the program prints the space character chr$(32). If the cell contains a "1" the program prints a block character: chr$(31).

After a cell is printed onto the screen, the data is transfered from the NE array (new) into the OL array (old). The generation variable GN is printed at the bottom of the screen and incremented by one.

When the screen has been displayed, the computer begins to calculate the next generation. It beings by checking the data in the OL array. Using the data in the OL array, the program creates the NE array.

First the eight neighboring cells are checked for each cell position. For each neighbor found, the variable N is incremented by one. When all neighboring cells have been checked, the outcome of the cell position depends upon the number held in variable N. (See also life rules; survival, death and reproduction)

What isn't stated explicitly in these program lines is the outcome of the cell when N = 2. When N = 2, the cell doesn't change in the next generation. If it contains an organism, the organism makes it to the next generation; if it's empty, it remains empty. The checking algorithm repeats for each cell in the array. Upon completion the program jumps to the display section to display the new generation and the process repeats.

Generations

By replacing the initial glider pattern, we can investigate more of life's little surprises. Replace the original program lines with the following program line to make a new starting pattern. This pattern creates an expanding population of organisms (Figures 7, 8, and 9).

```
REM  Enter Starting Pattern
NE(12,31) = 1
NE(12,32) = 1
NE(12,33) = 1
NE(13,30) = 1
NE(13,33) = 1
NE(14,29) = 1
NE(14,33) = 1
RETURN
```

Experiment with this program by changing the starting patterns as well as modifying the rules. You never know what you may come up with!

3D Life

The game of life has advanced beyond a two-dimensional game. Carter Bays, a computer scientist at the University of Carolina, has created three-dimensional versions of life. One version uses cubes held in three-dimensional space. A cube in *3D Life* has 26 neighbors instead of eight. The other version uses spheres. Unfortunately we don't have the space to explore these programs. Not to worry, for those interested in either version of 3D Life can receive more information by contacting Carter Bays at the University of Carolina.

Carter Bays Computer Science Department
The University of South Carolina

Columbia, SC 29208

Order Out of Chaos (Random Life)

This program is not as complex as the "Game of Life" but it generates very interesting screen images. It is a simple randomly initiated cellular automation. When the program is run, it fills the screen, line by line, with color pixels. When the screen is full, it begins to scroll upwards. Each new line generated on the screen represents a new generation. Pressing the "q" key exits the program, while pressing any other key randomly changes the rules upon which new generations are produced.

The keyboard is checked only once after each new line is printed onto the screen. This makes the program less than responsive to keyboard presses. That is a trade-off to have the basic program run as quickly as possible.

Order Out of Chaos—Basic Listing Explanation

This program uses a fastest algorithm than the Life program. Because of this, we can use pixels instead of chr$() characters, for a higher resolution display. Using pixels, however, slows the basic program considerably. I feel this is made up for by the resolution of CA pictures obtained from the program. Figures 10, 11 and 12 show the evolution of the cellular automation created by this program and captured on screen.

The program dimensions four arrays, OL(325) for old array, NE(325) for new array, KO(325) for color and code(16) for code.

The "code" array is filled with 16 binary numbers (1s or 0s). The program randomly changes these numbers upon either request of the programmer while the program is running or every time the screen scrolls upward. The automatic changing of the code mimics evolution. The code array may be considered the DNA of the program. Its sequence determines the next generation pixel. We will see exactly how this happens in a bit.

The user can change the code when the program is running by pressing any key except "q." The "q" key causes the program to end.

GOSUB "seed" jumps to a subroutine that fills the OL array with random sequence of binary 1s and 0s. The program reads through the OL array applying the "rule" line to create the next generation. Note that the first line of pixels is randomly created by the seed subroutine, just to get started, and it doesn't happen again. If you calculate all the possible variations in the Line "rule," you'll discover that it simply generates a number from 1 to 16 using the binary numbers in the OL array. The number is created by using five binary numbers from the previous generation (OL array); Y-1, Y, Y+1, Y+2 and Y+3. The number produced from RULE is used to pick a binary number from the CODE array. The binary number picked from the code array is placed in the NE array. The color for the character is also derived and stored in the KO() array.

When the entire NE array has been generated, it is displayed on the screen.

The next two lines allows the screen to scroll properly.

The next three lines accepts keyboard input. If you want to quit the

program, press "q." Pressing any other key brings the program to
subroutine

"change." This subroutine randomly changes the binary 0s and 1s held in the CODE array.

Modifying the CA Program

As the program is written, it uses a 320 x 200 screen with 16 colors. You can modify the program to use 32 colors or a higher 640 x 400 resolution screen.

The program automatically evolutes by going to the "change" subroutine every time the screen scrolls. You can stop the automatic evolution by changing one line:

```
IF ht = 184 then ht = 170: GOSUB change: LOCATE 23,1:PRIINT:PRINT
```

to

```
IF ht = 184 then ht = 170: LOCATE 23,1:PRINT:PRINT
```

If you change this line, the program will evolve only when the user presses keys as described previously.

The Dark Side of Cellular Automation

The dark side of cellular automation consists of computer worms and viruses. Computer worms and viruses are self-replicating programs that instigate themselves into computer operation systems. Viruses, like their biological counterparts, infect and replicate inside hosts. For computer viruses the hosts are programs files and diskettes.

Viruses are spread from computer to computer through "infected" diskettes or from a computer networks. Viruses that hide in the boot section of the computer’s hard disk or floppy diskette are called boot sector viruses. From this vantage point the virus attempts to spread itself by infecting any disks it has the occasion to corrupt.

The "Stoned" virus of IBM fame is an example of a boot sector virus. This virus destroys the directories and file allocation tables (FAT) of the hard drive while displaying the message "Your PC is stoned—LEGALIZE MARIJUANA." The file allocation table is a road map the computer writes on the disk telling it where it has stored all the files.

Another type of virus are file viruses. These viruses can either infect or attach themselves to program files. If the virus infects a programs, it writes viral code into the program itself, generally rendering the program inoperative.

A virus that attaches itself to a program waits until that program is executed and loads itself into memory along with the program. For instance, if you run a word processing program with an attached virus, the virus loads itself into the computer memory with the word processor. Once it's in memory, it stays there even when you're finished using the word processor. The virus stays in memory waiting for another program to be run. If you should run another program like a drawing program, the virus attaches itself to that program. This process continues until the virus has infected every executable file on your hard drive.

Worms are a class of viruses. These are usually found on mainframe computers. The most common attack of a worm is to replicate itself ad infinitum until the entire storage system of the computer is full of worm copies. As the worm eats up RAM and disk space, the computer begins to slow down drastically.

Bombs are viruses that are triggered by a date, time, or event. The Jerusalem-B virus goes off every Friday 13. Its main attack is erasing any program ran on that day.

Stealth viruses may be either boot or file viruses. It is called a stealth virus because it tries to hide from anti-viral programs. The manner by which the virus hides is ingenious; it changes the disk directory information to conceal its size and location.

Some computer scientist speculated that viruses are the first programs capable of existing without willful cooperation of humans. While this is true, it must be remembered that without humans to originally write the program code, viruses wouldn't exist at all.

Because of the potential damage that can be caused by computer viruses, the problem has created a market for anti-viral programs. There are many commercial anti-viral programs on the market that clean up and prevent viruses from attacking your computer system.

The National Computer Security Association has an eight-page document "Corporate Virus Prevention Policy" for sale. The document is targeted toward individuals who are responsible for their companies anti-virus policy. Cost is $9.95 for a printed copy, $12.95 for a disk with ASCII text file or $19.95 in *WordPerfect* format. Call 717-258-1816 for more information.

Core Wars

Core Wars are a recreational form of computer viruses. The game was created by A.K. Dewdney. The game is simple. Two programs are placed into a computer's memory. Each program’s job is to battle and annihilate the other program. The winning program takes over the allocated space in the computer’s memory.

The initial publication of the game was in the May 1984 issue of *Scientific American*'s "Computer Recreation" column by Dewdney. Since the initial description, the game gained so much popularity that tournaments are held for the best core war program.

Further information on Core Wars can be received by writing:

The Core War Newsletter
William R. Buckley, Editor
5712 Kern Drive
Huntington Beach CA 92649

International Core War Society
Mark Clarkson
8619 Wassall Street
Wichita, Kans. 67210

Software Available from:

Amiga PCs
Mark A Durham
8282 Cambridge Street # 507
Houston, TX 77054

Free Artifical Life Programs

If you belong to a large BBS network such as CompuServe, it is likely that there are artifical life programs there you may download for free. Check CompuServe’s Amiga File Finder.

The classic version of "Life" has multiple listings written with a variety of computer languages. You may want to download a compiled version for faster execution times.

Commerical Artifical Life Programs

We will end this chapter by noting a few commerical artifical life programs. These automation programs are available through standard distribution (stores) or directly from the manufacturer. I haven't personally used the *SimLife* program, but the reviews I have read on it are good.

CellPro
MegaGem
1903 Adria
Santa Maria, CA 93454
(805) 349-1104

SimLife
Maxis2 Theater Sq.
Suite 230
Orinda CA 94563
800-336-2947
510-254-9700