COMPUTER RECREATIONS

Sharks and fish wage an ecological war on the toroidal planet Wa-Tor

by A. K. Dewdney

Somewhere, in a direction that can only be called recreational at a distance limited only by one's programming prowess, the planet Wa-Tor swims among the stars. It is shaped like a torus, or doughnut, and is entirely covered with water. The two dominant denizens of Wa-Tor are sharks and fish, so called because these are the terrestrial creatures they most closely resemble. The sharks of Wa-Tor eat the fish and the fish of Wa-Tor seem always to be in plentiful supply.

This simpleminded ecology might appear stable, almost soporific, were it not for the fact that the shark and fish populations undergo violent oscillations. Many times in the past the fish population has been all but devoured, whereas at other times the sharks have starved almost to extinction (even when there were plenty of fish). Yet both sharks and fish survive. To discover why, I designed a program to simulate their feeding and breeding activities.

Before I had ever witnessed these ecological rhythms on a display screen, however [*see illustration on page 19*], I mused for a long time about the rules and the details of the WATOR program. Over lunch one day I found myself musing across the table from David Wiseman, who is my department's resident systems wizard at the University of Western Ontario. After describing the project to him I noticed that Magi (for such is Wiseman called) was smiling enigmatically. The next morning he proudly ushered me into his office to display a working program.

"Watch," he said and pressed a key. An initially random assortment of fish and sharks flickered slowly from point to point in what seemed to be a chaotic manner. Some sharks failed to eat and disappeared. Other sharks had offspring just as voracious as themselves. A few fish, lucky enough to occupy a region where there were currently no sharks, multiplied into a large school. Presently a number of sharks discovered the school, congregated at its edges and gulped their way a short distance into it. A few minutes later the summary of current statistics displayed on Magi's screen told the story: there were now 578 fish and just 68 sharks.

Someone walked into Magi's office and ran out again. Before five minutes had elapsed the room was crowded with people cheering on the sharks. Slowly a wall of sharks closed in on the hapless fish. Elsewhere on the screen a small school of fish slowly multiplied unnoticed. Groans went up when the large school of fish finally disappeared and sharks, dying one by one, milled about looking for prey. I thought of changing the rules to allow sharks to eat one another, but I realized that a feeding frenzy would not significantly prolong their existence and might put the early history of that other small school in jeopardy. When two roaming sharks finally stumbled onto it, the cycle began anew.

The program for Wa-Tor is neither very long nor difficult to write. Readers who have personal computers, even those with little programming experience, will find it a rewarding project when the code is finally written, debugged and running. Parameters such as breeding times, starvation periods and initial population sizes can be set before a run. It is then just a matter of sitting back and watching as an initially disorganized mélange of fish and sharks slowly forms ecological patterns.

The WATOR program embodies a number of simple rules that govern both shark and fish behavior. The creatures swim in a rectangular ocean grid whose opposite sides are identified in pairs. This means simply that if a fish or shark occupies any rightmost grid point and decides to swim east (to the right), it will reappear at the corresponding leftmost grid point. The same relation holds between the vertical extremes. The resulting two-dimensional wraparound space is really just a torus, the actual surface of Wa-Tor [*see illustration on page 20*]. Anyone writing his or her own WATOR program may select any convenient size for the ocean grid. For example, Magi, whose program runs on a VAX computer, has set up an ocean that is 80 points wide and 23 points high. My own version of WATOR, written for an IBM PC, uses a humbler, 32-by-14 ocean.

Time passes in discrete jumps, which I shall call chronons. During each chronon a fish or shark may move north, east, south or west to an adjacent point, provided the point is not already occupied by a member of its own species. A random-number generator makes the actual choice. For a fish the choice is simple: select one unoccupied adjacent point at random and move there. If all four adjacent points are occupied, the fish does not move. Since hunting for fish takes priority over mere movement, the rules for a shark are more complicated: from the adjacent points occupied by fish, select one at random, move there and devour the fish. If no fish are in the neighborhood, the shark moves just as a fish does, avoiding its fellow sharks.

The creator of WATOR selects five parameters in order to set up a given simulation. The parameters *nfish* and *nsharks* represent the numbers of fish and sharks at the beginning of a run. The program distributes the specified numbers of fish and sharks randomly and more or less uniformly across the planet's surface. The parameters *fbreed* and *sbreed* designate the number of chronons a fish and a shark respectively must exist before each has a single offspring. (Both species are apparently parthenogenic.) Finally, *starve* specifies the number of chronons a shark has in which to find food. If it swims about any longer than this without eating, it dies and sinks out of sight. During each chronon WATOR moves each fish and each shark once and displays the results on the screen. With rules no more complicated than these, one can watch the ecology of Wa-Tor lurching from crisis to crisis.

Magi and I have witnessed a number of five-parameter scenarios in which Wa-Tor's ocean became overpopulated with fish only to have the sharks eventually multiply to a point where all the fish were eaten and the sharks died. On other occasions we have seen all the fish in one large school being eaten. The sharks that had gorged themselves finally starved, never discovering a very small cluster of fish nearby. On a few occasions we have seen the prey-predator relation sustain itself through two or even three population cycles before the ultimate crash in shark population. Nothing in the parameters selected for those scenarios, however, gave any hint of the characteristics that would ensure an eternal ecology. How had the denizens of Wa-Tor survived?

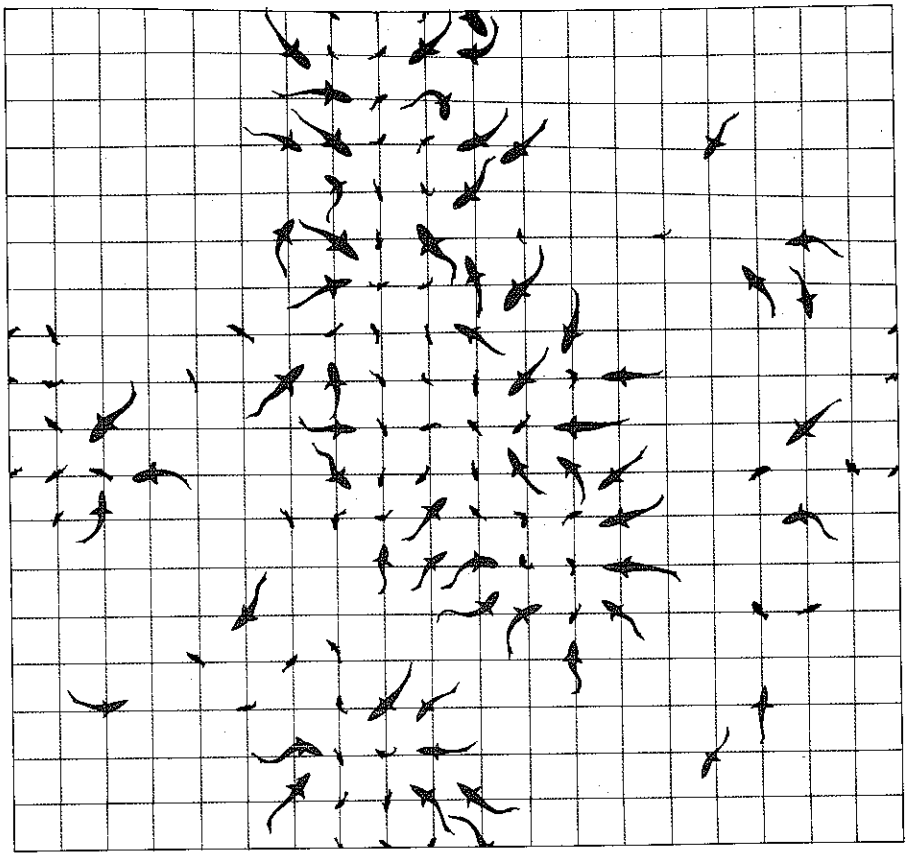
It has been said that biology is destiny. Magi and I are tempted to declare that ecology is geometry, at least as far as the planet Wa-Tor is concerned. The ultimate fate of a given scenario does not

seem to depend on the initial random distribution of a specified number of sharks and fish. Nor does it seem to depend in an accidental way on the actual random movement of sharks and fish. Instead the likelihood of a population crash appears to follow closely the fish-shark geometry that manifests itself on our screens: the more highly organized and localized either population becomes, the likelier it is that the ecology is doomed. Meditating on this theme, we were led to wonder how we might choose the five parameters in a way tending to break up the geometry. Then came a flash of insight: if sharks had congregated at the edges of a school of fish, one way to break up the resulting geometry would be to have the sharks breed less often. The congregation itself, after all, was less the result of motion than it was of breeding.

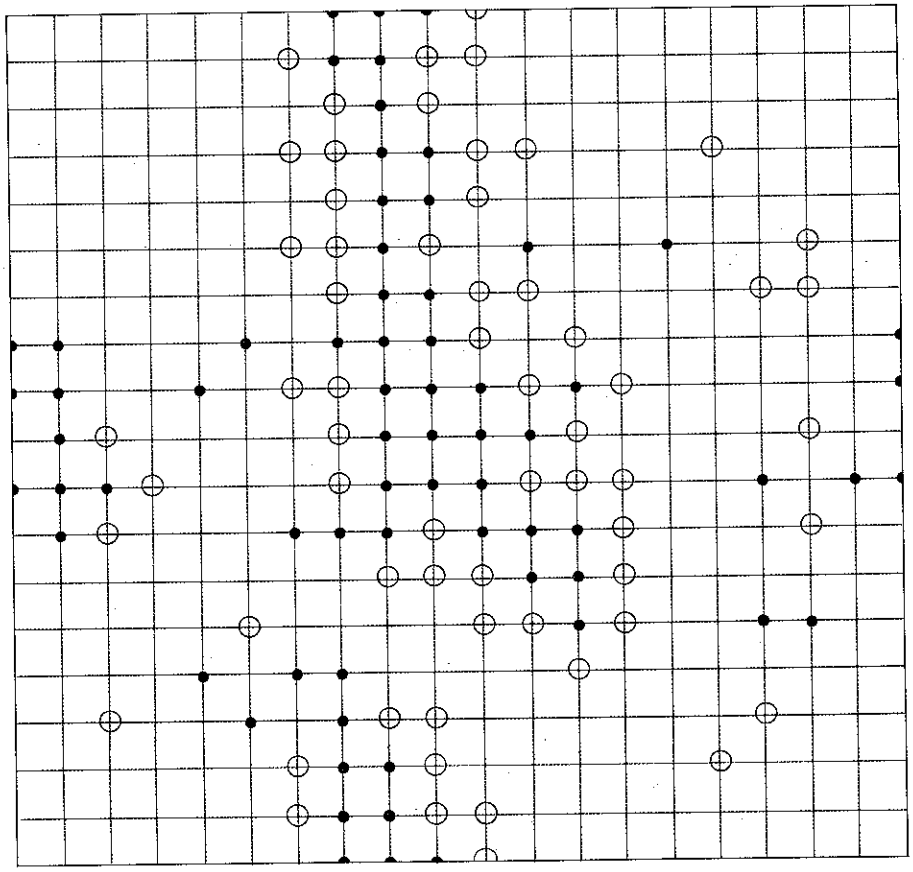
Before forming this hypothesis we had chosen roughly equal breeding times for sharks and fish. Balanced reproduction rates, we thought, would result in balanced populations. This kind of vague thinking probably accounts for many woes in today's technological world. In any event, I put 200 fish and 20 sharks in my 32-by-14 ocean and set the fish to breed every three chronons but barred any shark from reproducing before 10 chronons had elapsed. Shark starvation time was set more or less arbitrarily at three chronons. We were rewarded, after watching my rather slow program for 15 minutes, by seeing a full recovery from the initial population decline. Moreover, the geometry, although it was still present, was more suggestive than definite. Schools were shapeless conglomerations with ragged edges, and at some places on the screen sharks and fish milled about at random.

I let the program run all afternoon, glancing up occasionally from more important matters on my desk. The program ran all night, and when I visited my office after my morning lecture, I found fish and sharks still pursuing a cyclic existence. Here was Wa-Tor!

There are many ways to implement a WATOR program but perhaps the simplest involves a number of two-dimensional arrays. I use five arrays called FISH, SHARKS, FISHMOVE, SHARKMOVE and STARVE. These arrays, all 32 by 14, keep track of the positions and ages of sharks and fish. Specifically, FISH(I,J) represents the presence or absence of a fish at the point with coordinates (I,J). If a fish is absent, the position has the value -1. Otherwise it contains a record of the age in chronons of the fish that is present. The same scheme is used for the array SHARKS to keep track of the positions and ages of the sharks. The array FISHMOVE holds a record at each position of whether a fish has been moved there during the current computational cycle.



A realistic view of sharks eating fish



A more easily programmed view, in which circles represent sharks and dots represent fish

Such a record enables the program to avoid moving a fish twice during the same chronon. SHARKMOVE fulfills the same function for sharks. The array called STARVE registers the time at which a shark last ate. If there is no shark at a position, the entry is −1.

The simplest display of the action on Wa-Tor is a line of characters on the screen for each row in the arrays; a blank at a position means it is unoccupied. A period (.) represents a fish and a zero (0) represents a shark. Although this display might seem to be limited, it is surprisingly informative and enjoyable to watch.

In WATOR's initial phase the required numbers of fish and sharks are scattered uniformly over the toroidal ocean. The program then cycles through the three segments or subprograms described below; each program cycle occurs during one chronon of time.

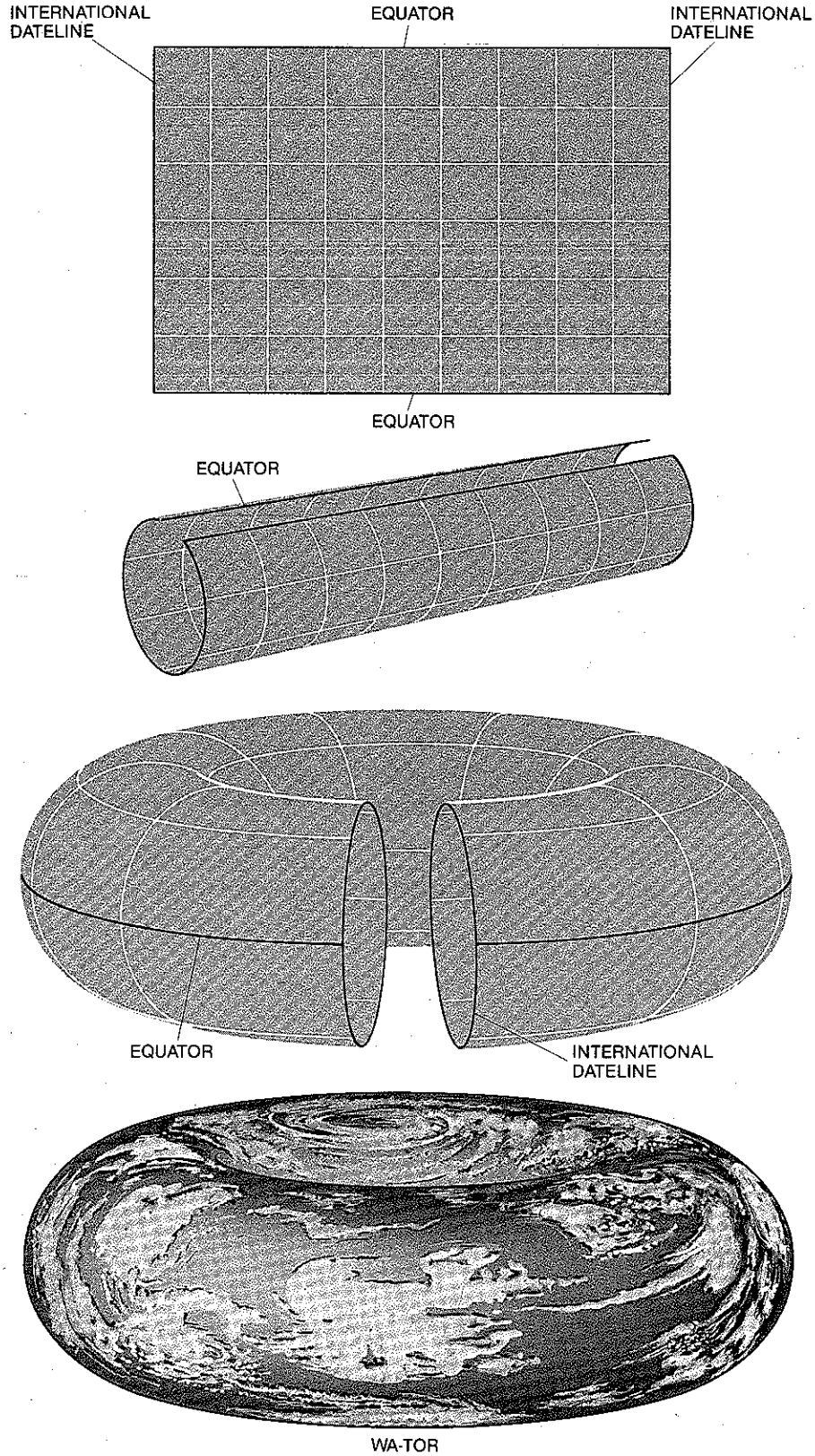
FISH SWIM AND BREED:
For each fish in the FISH array, the program makes a list of adjacent unoccupied positions and moves the fish to one of these at random. This means FISH must be set to −1 at the old position and set to the fish's current age at the new position. The array FISHMOVE is updated in the manner described above. If the fish's age equals *fbreed*, the program puts a new fish at the old position and gives age 0 to both fish. Again FISHMOVE records the new fish. If all adjacent positions are occupied, the fish does not move or breed.

SHARKS HUNT AND BREED:
For each shark in the SHARK array, the program makes a list of adjacent fish positions (if any). The shark chooses one of these at random, moves there and eats the fish. This means not only that the program must modify SHARKS and SHARKMOVE as it modified FISH and FISHMOVE, but also that it must set the corresponding position in the FISH array to −1. Also, STARVE at that position is set to 0. If there are no adjacent fish, the shark moves just as a fish does. If the shark's age equals *sbreed*, a new shark is produced in exactly the same way as a new fish is.

DISPLAY:
The program scans both the FISH array and the SHARKS array. It displays a period for each fish and a 0 for each shark. The display can be done all at once in this way or broken into two parts: one executed after the fish have moved, the other executed after the sharks have moved.

To populate the initial ocean, the programmer constructs a loop that generates two random numbers *nfish* times. The numbers are scaled to the horizontal and vertical dimensions of the ocean he or she intends to have. At each of the random positions thus selected, the program places a fish in the FISH array and assigns it a random age between 0 and *fbreed*. Sharks are distributed similarly. In both cases the position is checked to see if it is already occupied. The effect of giving both sharks and fish random ages is that they then breed at random times in a natural way. Without this precaution one would witness the sharks and fish suddenly doubling in numbers, a disconcerting and unnatural sight.

There may be novice programmers



The toroidal planet Wa-Tor and its representation on a flat map (or a flat computer screen)

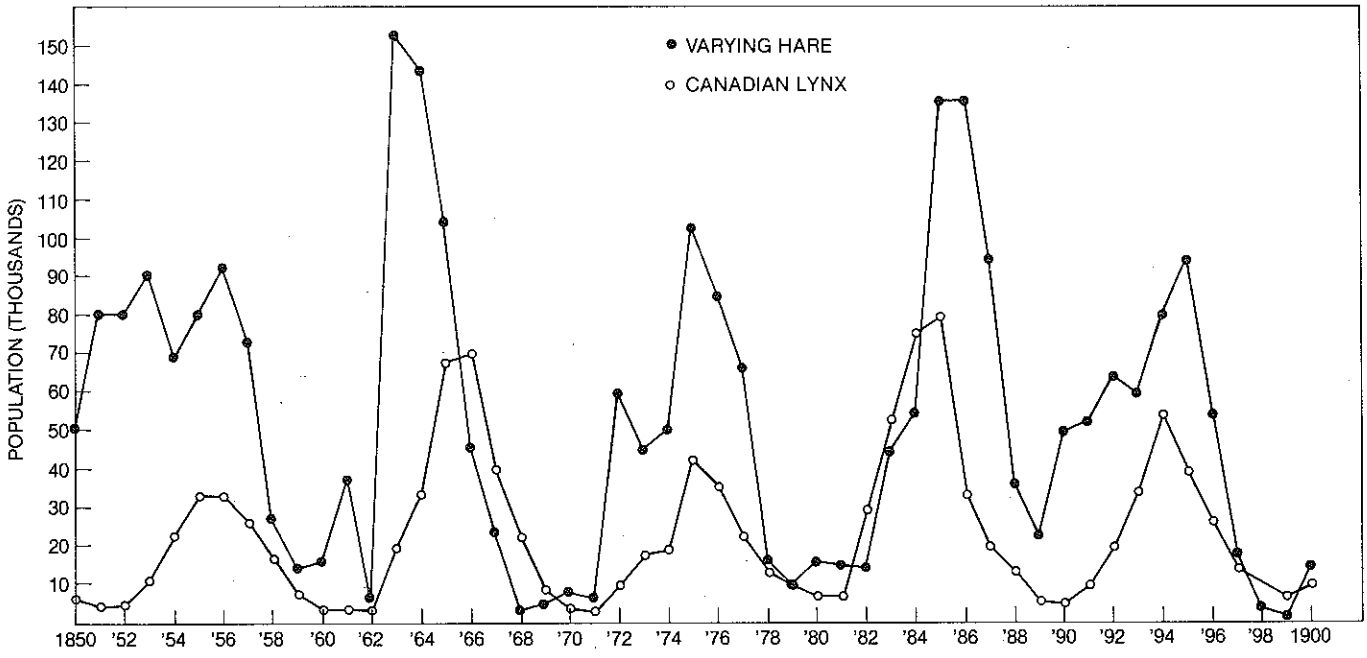
who find the foregoing description a bit too general to form any clear idea of how to write a WATOR program. Those programmers can begin by writing what is known as a staggering-drunk program. Such a program might consist of a single loop (say a while-loop) that has seven instructions. These are written in nonprejudicial algorithmic language. Assignments are indicated by left arrows and the variables X and Y are the coordinates of a staggering drunk. They are altered according to the random integer assigned to a variable *direction*. Depending on whether this integer equals 0, 1, 2 or 3, the drunk (a point on the screen) moves north, east, south or west.

```
direction ← integer part
  of (random × 4)
if direction = 0 then X ← X + 1
if direction = 1 then X ← X − 1
if direction = 2 then Y ← Y + 1
if direction = 3 then Y ← Y − 1
display (X, Y)
```

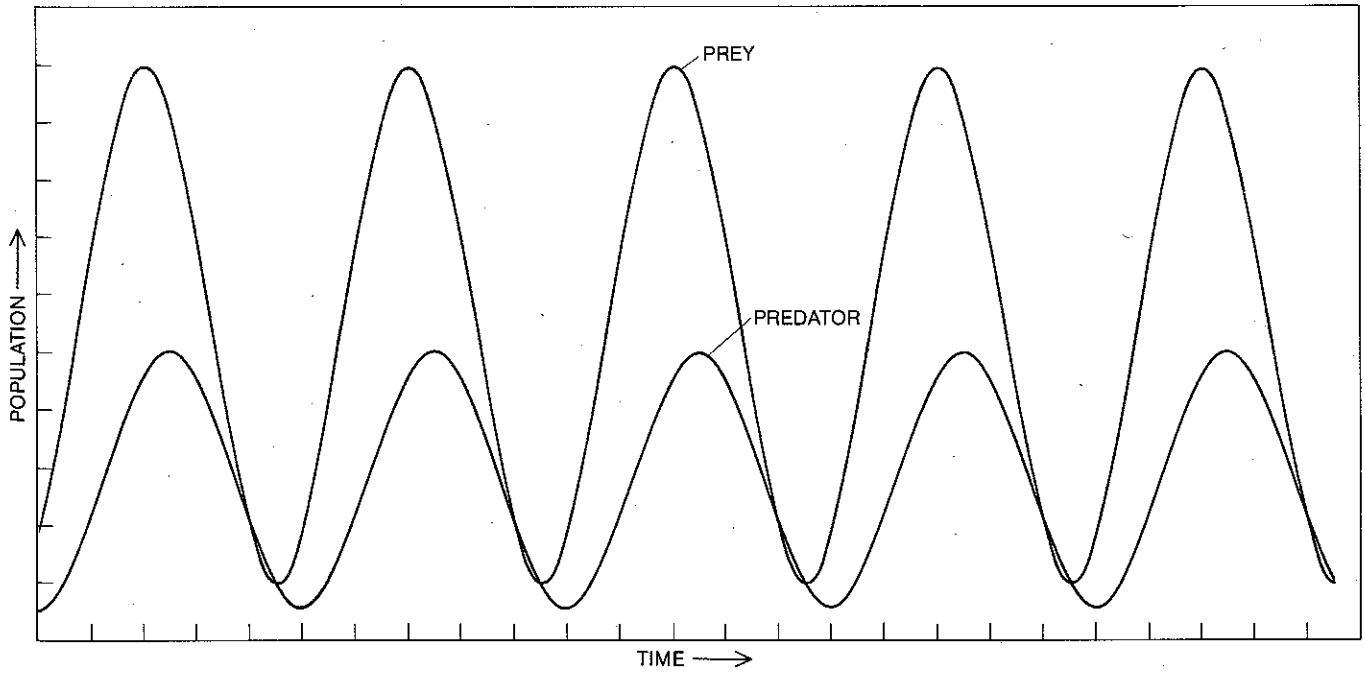
If your particular random-number generator produces a decimal number *random* between 0 and 1, this algorithm will scale it to a decimal number the value of which lies between 0 and .3999. The integer part of the resulting number must be 0, 1, 2 or 3.

I cannot claim that watching a point of light wandering minutely on your screen matches the ecological drama of the sharks and fish, but writing this program does give some insight into how parts of WATOR might be constructed.

Expert programmers reading this column will have thought of other approaches to writing the WATOR program. The amount of processing can be greatly reduced by using linked lists to



Numbers of lynxes and hares (in units of 1,000) trapped for the Hudson's Bay Company from 1850 to 1900



A theoretical predator-prey relation: a solution to the Lotke-Volterra equations

keep track of sharks and fish. With such a data structure the time required for one computational cycle is proportional to the number of sharks and fish present and not to the size of the ocean.

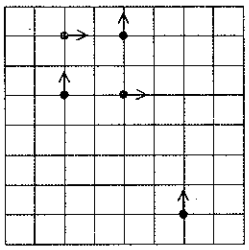
WATOR may yield some insights into animal populations here on earth. We know that small populations face a high probability of extinction and, even if neither predators nor prey die off, they are almost certain to undergo cyclic changes in number. In simple predator-prey ecosystems the predator and prey populations sometimes follow two overlapping cycles of population maxima and minima. The sizes of the populations of the varying hare and the Canadian lynx recorded by the Hudson's Bay Company from 1847 to 1903 in the Canadian subarctic follow this pattern [*see illustration on preceding page*]. The figures give the number of each species trapped from one year to the next. Presumably these numbers are proportional to the actual population sizes present during this period. If they are, the cycles are easily explained as the result of lynxes eating their way into an ever increasing hare population that begins to decline as the number of lynxes increases. Soon there is less food for the lynxes and they begin to starve, breed less or both. When the lynxes are reduced in numbers, the hares begin once again to multiply.

Contrasted with this chart is a smooth set of curves representing a solution to the Lotke-Volterra equations. These equations were first formulated in 1931 by V. Volterra, an Italian mathematician. They assume what might be called a continuous predator continuously in search of a continuous prey. The solutions to these equations exhibit a cyclic variation that, at first glance, appears to reflect the lynx-hare empirical data. Biologists are not in agreement, however, that the lynx-hare numbers are explicable by such simple reasoning. For one thing, at least two other predators of hares are involved: microbes and man.

It makes perfectly good sense, however, to compile statistics on the sharks and fish of Wa-Tor, and Magi and I have done so. Our recent graphs of the shark and fish populations tend to look more like the lynx-hare charts than the Lotke-Volterra solutions do. Still, we continue to be puzzled by the long-term instability shown by certain parameter combinations. Perhaps some reader, working with his or her own WATOR program, will provide further insight. Is there some kind of general rule we might use to predict, for a given combination of parameters, whether the resulting ecology will be stable? To what extent do the cyclic fluctuations follow the Lotke-Volterra equations?

The ocean of Wa-Tor is toroidal for a very simple reason: it is much easier to write a program for an ocean that has no boundary or shore. If the ocean is to be, say, 32 units wide, it is a simple matter to use numbers modulo 32 as the X coordinates of fish and sharks. If they have X coordinate 31 and appear on the right-hand side of the screen during one chronon, they may well have X coordinate $32 = 0$ and appear on the left side during the next chronon. The same system is used vertically.

The toroidal ocean of Wa-Tor gives rise to some very strange effects, as exemplified by the following puzzles. The first of these effects involves a bug in an early version of my WATOR program. This bug caused each fish to swim one unit north and each shark to swim one unit east during each chronon of time. Thus a shark got to eat a fish only if it found itself occupying the same location as its prey. In the ocean below, how many fish were never eaten by sharks?



Another puzzle involves intelligent sharks and fish. Suppose each shark and each fish takes turns moving to any of its four neighborhood points. It turns out that a single fish, if it is intelligent enough, can always evade a single shark, no matter how intelligent the predator. In the toroidal ocean of Wa-Tor, two sharks hunting a lone fish may produce a different ending. If you endow each creature with all the intelligence you like, even allowing the sharks to hunt cooperatively, can you discover a way out for the fish? The result does not depend on the dimensions of the ocean.

The subject of perceptrons ["Computer Recreations," September] reminded some readers of applications and spurred others to investigate the subject on their own. Ed Manning of Stratford, Conn., built a "perceptron of sorts" 10 years ago designed to convert real images into the digitized squares of a perceptron's retina. Manning was one of a few people who noticed a mistake in the multiple-rectangle window perceptron shown at the top of page 34 of the September issue: the last four demon patterns should each be half blue and half white. Manning wondered whether the mistake was "intentional to plumb readership." I am tempted to say that it was.

Gary D. Stormo, an investigator in the Department of Molecular, Cellular and Developmental Biology at the University of Colorado at Boulder, has used the perceptron concept in automated pattern recognition. Specifically, he has constructed a perceptron-weighting function to recognize binding sites in messenger-RNA nucleotide sequences. He uses the perceptron convergence theorem to guide the performance of his perceptron toward an optimal level. The results have been very encouraging: the perceptron recognizes binding sites with "substantial success."

Any window perceptron that includes either an all-white or an all-black window pattern in its list is a good perceptron. Lowell Hill of Venice, Calif., noted this and wondered whether an all-white or an all-black retina constitutes a legitimate picture. The answer depends on the pattern. It seems reasonable, in the case of the multiple-rectangle perceptron, to regard an all-black retina simply as one large rectangle.

In the course of a most successful foray into the mini research project I suggested, Constantine Roussos of the Lynchburg College Computer Center in Lynchburg, Va., decided to exclude window perceptrons with all-white or all-black window patterns. Among his achievements is a characterization of good perceptrons (those that recognize at least one pattern). The characterization uses translational relations between the window patterns on the perceptron's list. If one shifts a window pattern by a single unit in any of the four principal directions, one must obtain another window pattern on the list. Roussos then concentrated on minimal window perceptrons, those with a list that cannot be further reduced without destroying the goodness of the perceptron. Such perceptrons are building blocks for the set of all good perceptrons. Roussos wrote a computer program that discovered all minimal window perceptrons having list sizes of orders 2 through 5. No minimal window perceptron of order 6 exists. Roussos raises a challenge by turning around the task I had set: I proposed that readers find a pattern recognized by a given perceptron; Roussos suggests discovering a perceptron that recognizes a given pattern.

John M. Evans of Hartford, Conn., blames perceptron failings on the restriction inherent in a two-level hierarchy of local demons reporting to a single head demon. By introducing a kind of demonic middle management, Evans overcomes the connectivity limitations for ordinary perceptrons discovered by Minsky and Papert. The low-level demons themselves constitute a kind of retina whose blacks and whites correspond to whether particular demons report or not. A second layer of demons watches the pattern created by the low-level demons; it reports the presence of subpatterns to the head demon. A three-layer device can distinguish which of the four test patterns are connected and which are not.