The CAM/PC exerciser CAMEX

Harold V. McIntosh
Departamento de Aplicación de Microcomputadoras,
Instituto de Ciencias, Universidad Autónoma de Puebla,
Apartado postal 461, 72000 Puebla, Puebla, México.
E-mail:mcintosh@servidor.unam.mx

Jun 21, 1992

Abstract

A very general commentary on the program CAMEX, accompanied by an assortment of experiments which can be performed using the CAM/PC board. These include the use of one, two, and three dimensional automata, although the primary utility of the board lies with two dimensional automata evolving on a 256x256 screen. Binary Moore and von Neumann neighborhoods are possible in each of the two half-CAM's, as well as four-state von Neumann neighborhoods; many additional mixtures are possible.

Contents

1 LCAU program set

Over the course of several years a collection of cellular automaton programs has evolved for use in a course at the Universidad Autónoma de Puebla entitled FORTRAN III, dedicated to graphical techniques. Besides a display of automaton evolution, the programs feature the calculation of de Bruijn diagrams (from which the periodic aspects of the evolution can be deduced), an analysis of equilibrium probabilities, and calculating ancestors.

Because of the amount of computation involved, the program set has emphasized one dimensional automata in preference to two or three dimensions. With the availability of a video controller, such as the Automatrix CAM/PC, capable of updating a whole planar array of cells while scanning the screen, more ambitious studies of bidimensional automata can be undertaken, toward which CAMEX represents an initial step.

CAMEX follows the layout of the LCAU programs, without emphasizing any single neighborhood or state set; whilst LCAU22 was dedicated to binary one dimensional automata with second neighbors, for example. After an initial sign-on message (which actually initialized a random number generator), there was a main menu which permitted excursions into a variety of submenus; all the menus were activated by single keystrokes. Sometimes additional symbols would follow the initial key, but a numerical prefix never generated multiple execution.

Uniformity between menus was sought, to make the significance of the keys easier to remember; the symbols ? and ! often exhibited a rather terse help panel. Only recently have the programs begun to incorporate a systematic usage of function keys, mouse movements, or even pop-up menus. However, a recognition of the general layout will assist the user in working through an otherwise rather barren environment.

With respect to LCAU, the collection should be consulted for automata which lie outside the range covered by CAMEX, and even to get a much more elaborate treatment of those lying within a common range. The reason for the first recomendation is fairly obvious, because the range of states and lengths of neighborhood to which CAMEX is adapted is a function of the CAM hardware; the best results are surely to be obtained when the software is compatible with the hardware.

Within the common range, the difficulty results from an historical anomaly which has to do with other factors entirely; the change of the design team between the Intel 8085 and the Intel 8086. In the process, no provision was made for detecting a carry originating from the overflow of segment addresses, limiting the size of arrays that could be used without a complicated addressing scheme.

The practical consequence is that CAMEX's data space has to be apportioned with care, lest CAMEX itself become more complicated than presently convenient; therefore CAMEX lacks those portions of LCAU which might have been copied bodily. Instead, the space has been devoted to a better treatment of the same aspects of two dimensional automata, the real province of the CAMs. Even without the competition from one dimensional automata, the limitations of the architecture of the Intel 8086 and its successors are acutely apparent.

For example, de Bruijn diagrams for one dimensional automata can be calculated for three or four generations; for even as many as a dozen in the case of the simpler automata. The same limitations preclude even two generations in two dimensions and exclude three altogether.

2 CAMEX main menu

CAMEX works with the two principal two-dimensional neighborhoods, Moore and von Neumann, plus several of their variants, but so far does not entertain Margolis neighborhoods.

Since full neighborhoods are tedious to describe, generators following such concise rules as totalistic, semittotalistic, or even-odd-center are provided. Each variant has a submenu within which its own rule can be edited.

2.1 General layout

There are explicit generators for renowned automata such as as *Life*, *WireWorld*, Bank's rule, Fredkin's XOR rule, or the Zhabotinsky reaction.

Additionally, bitplanes for the Moore (2,1) neighborhood and the (4,1) von Neumann neighborhood can be edited. Much work remains to be done to facilitate bitplane editing, especially with the help of a mouse. Up to the present, the area of editing Margolus planes has been completely untouched.

For many applications, simple patterns or random fields provide good initial configurations. CAMEX has instructions that will create various patterns, with densities which are parameters which the user can set to preferred values. No doubt as time passes and experience accumulates, and especially as a repertoire of basic techniques accumulates, the generation of sample bitplanes will become simultaneously more concrete and more varied.

Naturally, constant fields can be provided; there are some other designs, such as checkerboards, with their own parameters. Besides that, shifting and permutation of the bitplanes can be performed.

Some of the submenus contain provisions for loading prepared bitplanes from disk files, and also to save them from time to time while running CAMEX. Insofar as these facilities exist, they are discussed separately, in the context of their own submenus.

The layout of the main menu is shown in the accompanying diagram, Figure 1.

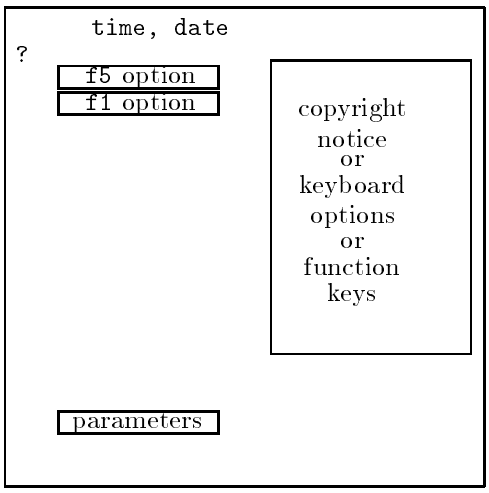

Figure 1: CAMEX main menu

2.2 Copyright notice

The copyright notice, which is only visible on startup, is displayed in a panel at the right side of the screen, the same area which is also used for the two help panels. One of them contains a terse list of the keyboard options available in the main menu, summoned by typing `?`; the other explains the use of the function keys, summoned by `!`.

This style of describing all the options in two groups via the same two keys, is followed throughout all the submenus, mainly because screen space is insufficient to place them all together.

The text of this program identifier is shown

in Figure 2, in which the current version number is visible; a number which consists of the date of issue, rather than the composite of version and revision which is often seen attached to programs.

```
"---------------------------",
"---        CAMEX        ---",
"---   (CAM Exerciser)   ---",
"---------------------------",
"A collection of simple test",
" operations  demonstrating ",
" some  capabilities of the ",
" Automatrix  CAM/PC  board.",
"---------------------------",
"                           ",
"     Harold V. McIntosh    ",
"    Apartado Postal 461,   ",
"   (72000) Puebla, Puebla  ",
"          MEXICO           ",
"                           ",
"       June 6, 1992        ",
"                           ",
"  ! for function key menu  ",
"  ? for keyboard options   ",
```

Figure 2: Copyright notice

In point of fact, issues of copyright and possible patent issues have probably reached extremes which they may not merit. Programs of proven commercial value invite conflicts and abuses most likely to remain remote from programs written for scientific purposes or for individual entertainment; in any event "intellectual property" is supposed to be born with an inherent copyright which only needs to be registered if and when the need arises.

The shrink-wrap contract which accompanies Turbo-C compilers mentions that passing along a valid copyright notice is a condition for incorporating portions of the Turbo-C library in ones own executable programs; all in all, a fair request.

Most people who place programs in the "public domain" are not really doing so; it is one thing to condone limited copying and sharing of a program, but quite another to forsake the consequences of some future popularity, or of some later version having greater attractiveness and economic potential. Having adequate acknowledgement of the source, and the retention of commercial rights, are realistic expectations for an author of modest programs; in any event that is the case with CAMEX.

The source code for CAMEX contains more explicit historical information, including the dates of first creation of the modules and the dates of major revisions to each. Attention should be given to possible discrepancies arising from differences in the dates of a given copy of the program and of the instruction manual, since their revisions are likely to occur at different times and independently of one another.

2.3 Main keyboard menu

The main menu now contains the following items (which are subject to change from time to time):

S,s - Run on, single step,
q - return to DOS,
z#,u# - clear,set bitplanes,
p#,R#,c#,w# - random points,
H#,h# - make checkerboard
(# = hex bitplane select'n),
y - random ellipse plane 0,
vx - permute the planes
(x=0,1,2,3,C,c,X,x,l),
d - de Bruijn diagram,
r - edit rule (see f1),
l - edit bitplane (see f1),

n# (#=nsew) shift plane 0,
N# (#=nsew) shift all pl,
f1 - selects rule type; rule
can be edited with r, plane
with l (if appropriate),
others — see source code.

Details of the options are provided in the following paragraphs.

2.3.1 Run the CAM

In general terms, `s` and `S` are used throughout the program to run through a cycle of evolution by the CAM. Single step operation results from typing `s`, whereas `S` will run on and on until another key is pressed. Insofar as possible, these letters are respected throughout all the submenus.

2.3.2 Return to DOS

Return to the DOS operating results from typing `q`, whereas the return from subroutines at all levels results from TYPING CARRIAGE RETURN. This difference prevents one from accidentally exiting from the program through an excess of enthusiasm when leaving a submenu. It is probably not surprising to be told that typing sequences of symbols frequently becomes reflexive, resulting in exasperating errors when the sequence has to be varied slightly, or has been altered unexpectedly.

2.3.3 Bitplanes

Many of the options in the main menu relate to setting up standard bitplanes. Aside from setting or clearing them individually or collectively, they can be seeded with random points according to four densities. These are high, medium, low, and sparse; in each case the actual number is a variable which can be set by the parameter options.

Probabilities are all given in mils; n per thousand, making 500 correspond to 50% probability, or equal likelihood. Sparse refers to a fixed number altogether, whose locations are chosen at random.

Often it is desired that only a small portion of a plane be occupied, allowing the expansive properties of a rule to be judged. To produce this contrast, the random points can be confined to the interior of a slightly skew ellipse, whose radius and density are also parameters.

Checkerboards with squares of different sizes represent another useful variant on standard initial configurations; provision also exists for setting them up.

The design of the CAM hardware allows the possibility of writing to multiple planes, but confines reading to a single plane. Following the hardware rigorously results in operations on the plane having a hexadecimal parameter ranging between `0` and `f` while writing, but a decimal parameter lying between 0 and 3 for reading. The hexadecimal parameters are microprogrammed in the obvious way.

It is possible to adjust the planes and to permute them, but so far these options have not been worked out with the full care which they deserve, and are still likely to vary from one version to another.

2.3.4 Shifting

Planes can be shifted by `n` (plane 0) or `N` (all planes) with directional arguments, but that just loads an appropriate rule table; `s` or `S` have to be used to do the actual shifting. There is an inconvenience that one loses the current rule table in the process, making it necessary to reload it. Sometimes one doesn't know, or has forgotten; sometimes pressing INSERT again suffices.

Shifting the planes is most useful on

startup, which typically loses the origin of the bitplanes, making it desirable to center a previously existing image once again. Images can be carried over, from one session of CAMEX to another, or even from a previous usage of the AUTOMATRIX FORTH program. The principal requirement is that the computer should not have been reset nor turned off during the interim.

2.3.5 Permuting

The option `v` will exchange planes, exchange and complement, exchange half-CAM's, and so on. The exact list is:

0 - complement plane 0
1 - exchange planes 0, 1
2 - exchange planes 0, 2
3 - exchange planes 0, 3
c - cyclic permutation
C - anticyclic permutation
x - exchange evens, odds
X - exchange CAM-A, CAM-B
l - longer cycle: "Dabc"

As more and more demonstrations and specialized submenus have been added, the use of this option has declined.

2.4 Function keys

The main use of the function keys is to expose additional menus; they are all described more fully in their own section. Two, `f9` and `f10`, are used to manipulate the color palette and to single out individual bitplanes. ERASE is used throughout CAMEX to clear the CAM planes, INSERT and `f5` to initiate processes.

The process may be nothing more than to load the CAM's rule table; in any event there are two lines in the upper left hand corner of the screen which reveal the process. Normally the lines are inactive; returning from the menu search which pertains to them will activate them. So also will the page and paragraph keys, showing one line at a time on the console. The CONTROL options move them to their respective extremes, in harmony with similar usage in text editors and similar programs.

In a similar fashion, the horizontal arrow keys will illuminate the parameter panel in the lower left hand corner; the vertical arrows raise and lower the parameter values. The arrows are always active, even when no display is showing; which may not have been a good decision given the possibility of inadvertently hitting the keys or moving the mouse.

2.5 Editing tables and planes

Originally the main menu was a center of operations from which a variety of initial conditions for a fixed rule could be set up and their evolution watched. Alternatively, one of a number of demonstrations could be chosen and set in motion. This is still a significant style of operation, but the number of rule configurations has grown, often bringing with them a specialized submenu of their own.

This increase in versatility has given fundamental importance to the options `r` and `l`, which permit editing the rules, or editing the bitplanes, respectively. In turn, this has created a tendency to repeat many of the operations of the main menu over again within each of the submenus. The increase in the total length of CAMEX is more than compensated by the increase in user convenience.

The particular rule, or the particular viewpoint toward the planes, is established via one of the other menus, typically the one accessible under `f1`; Conway's *Life* is the default choice, as it is always an interesting automaton to work with.

3 CAMEX function keys

Besides the main menu, whose contents can be displayed by typing `?`, there is a function key menu which responds to `!`, revealing several prepared options which can be used as demonstrations, or as the initial point in a further investigation.

f1 - rule or bitplane menu,
f2 - rec demonstrations,
f3 - edit rec expression,
f4 - execute f3 expression,
f5 - execute f2 expression,
f7 - parameters & values,
f8 - alternate INSERT,
f9 - show bitplane,
f10 - change color palette,
^ X - go to extreme X value,
X - reverse X's direction,
insert - install f1 option,
erase - clear all planes,
up/down - f1 sequence (ins),
page u/d - f2 sequence (f5),
arrow l/r - parameter seq,
arrow u/d - incr/decr par,
? - keyboard options.

3.1 The automaton menu

One collection (`f1`) consists of some typical rules and initial configurations, which can be set up (but not executed) by using the INSERT key (this allows browsing without making any selection). The keys `s` and `S` put the board in motion, `S` can be stopped by any subsequent keypress.

1 Conway's Life,
2 reversible Life-based,
3 reverse of reversible,
4 Life on a checkerboard,
5 One dimensional (2,1),
6 One dimensional (4,1),
7 Points generate axes,
8 totalistic (2,1) Moore,
9 totalistic (4,1) von N,
10 cyclic eater (4,1) v.N,
11 Zhabotinsky (4,1) v.N,
12 Banks' rule (2,1) v.N,
13 scurrying bubbles (M),
14 Silverman (Moore&echo),
15 random (2,1) Moore,
16 random (4,1) von N,
17 semitotalistic Moore,
18 semitotalistic von N,
19 even-odd-center (Moore),
20 even-odd-center (von N),
21 ellipse in plane 0,
22 few points in plane 0,
23 sparse random plane 0,
24 (4,1/2) plane Quad (G),
25 symmetric (2,1) Hex,
26 (16,0) in all planes,
27 (2,1) von N in plane 0,
28 net tracer in plane 0,
29 Zhabotinsky-like rule,
30 One dimensional (3,1),
31 Totalistic (2,1) Hex,
32 s.Totalistic (2,1) Hex,
33 e-o-c (2,1) Hex,
34 (2,1/2) plane Quad,
35 3D (2,1) von Neumann t,
36 3D (2,1) von Neumann st,
37 edit CAM color map,
38 WireWorld.

3.2 The REC menu

A second collection (`f2`) consists of packaged demonstrations, written in the programming language REC, many of which run through a series of steps. Their operation is launched by the key `f5`; the programs can be edited with the help of `f3` and executed by `f4` by a user who knows or can deduce the REC language, but this is not intended to be necessary to enjoy the demonstrations.

On the other hand, the REC demonstration menu consists of the following items:

1 null program .
2 free running evolution .
3 keep clearing plane 1 .
4 reversible and reverse .
5 Life on a checkerboard .
6 Life with glider trace .
7 Life with simple trace .
8 diag reflect pl 0, 1 .
9 adiag reflect pl 0, 1 .
10 horiz reflect pl 0, 1 .
11 vert reflect pl 0, 1 .
12 rotate by shearing .
13 rotate backwards .
14 .
15 interesting eater cycle.
16 Silverman rule .
17 Life glider demo .
18 spiral with bubbles .
19 gridwork from points .
20 two-color Life .

3.3 The parameter menu

Another collection contains the parameter options `f7`, which can be chosen by moving the cursor. Single line extracts from these three panels can be viewed by using the cursor arrows or the page and paragraph keys, just as though one were using one of the popular text editors. Finally, keys `f9` and `f10` govern the color palette; this allows individual bitplanes to be viewed and agreeable (or even disagreeable) colors to be chosen.

1: atern - alternate rule
2: wfrno - Wolfram rule number
3: hdens - high density - mils
4: mdens - medium density - mils
5: ldens - low density - mils
6: wdens - scarce dots - number
7: elrad - radius random ellipse
8: nchek - ch. squares per row
9: tacol - table color
10: macol - marker color
11: cucol - cursor color

In greater detail, the significance of each parameter is the following.

1: atern - Chooses alternate lookup tables for the Moore neighborhoods when they exist; range 0-4.

2: wfrno - A number in the range 0-255 which is the Wolfram rule number for the one dimensional binary automaton option.

3: hdens - The density (mils) of live cells for the option `c`.

4: mdens - The density (mils) of live cells for the option `R`.

5: ldens - The density (mils) of live cells for the option `p`.

6: wdens - The actual number of dots for the option `w`.

7: elrad - The radius (in units of about 6 pixels) of the ellipse for option `y`; range 0-54.

8: nchek - The number of checkerboard squares for option `k`.

9: tacol - The number in the range 0-255 used by `videocattr` to display table entries; it combines foreground and background.

10: macol - The number in the range 0-255 used by `videocattr` to mark table entries; it combines foreground and background.

11: cucol - The number in the range 0-255 used by `videocattr` as a cursor when displaying tables; it combines foreground and background.

4 Future developments

Even as CAMEX is under development, directions can be seen in which it could profitably be extended.

4.1 Manual

In partial atonement for the lack of an operating manual, the C language source code for CAMEX is provided. Even persons unfamiliar with that language can scan the menus in search of options, using the accompanying comments as a guide to their meaning; experimenting with the keys thereby discovered cannot cause serious damage. Evidently an actual manual is high on the list of priorities.

4.2 VGA controller

The use of a CGA video controller to accompany the CAM/PC board strongly affects the possibilities for the probability plots; the next level of controller would even double the admissible text, allowing the full von Neumann (4,1) table to be shown, as is done for the Moore (2,1) table. If higher density controllers could be assumed, better presentations would be possible.

4.3 Mouse

Convenient as a mouse may be, it is not really essential to most programs; but its importance for editing plane areas cannot be denied. More facilities can be expected in future versions of the program.

4.4 Sample rules and bitplanes

There is a tremendous lore of automata, especially concerning *Life*, which will gradually be worked into the program, especially with the assistance of the REC compiler which is already present.

4.5 Margolus neighborhood

Inasmuch as the Margolis neighborhood is an important feature of the CAM/PC board, future options will doubtless allow its rule set to be edited also.

4.6 Intel 8086 CPU

As work with CAMEX has progressed, various degrees of progress have been made toward realizing the objectives outlined above, just as some additional design considerations have been exposed.

The existence of the `64K` segment barrier in Intel products has been especially aggravating, although increasing CPU speed in the more recent members of the series has tended to conceal the consequences of this design defect. It is no less annoying that exotic pointer models would have to be used in C, a language chosen for portability, to remedy a situation which never should have occurred.

This is not to say that the discipline of minimizing data space has not had some beneficial side effects, but good programming discipline should have led to the same results independently of pressure from this source.

4.7 Bitplane editing

As automata such as *WireWorld*, *Life*, or even the Zhabotinsky reactions bave become established in CAMEX, they have called for larger scale bitplane editing than is presently available. Other automaton programs have some excellent editing facilities, which should eliminate any doubt that improvement is possible.

5 The (16,0) automata

Because of the exceptional role given to central cells in the CAM boards, one way to make all four planes interact with one another is to assume a neighborhood of zero radius. Although the resulting automata are rather trivial (but a goodly number to choose from), they provide an eminently practical way of copying or permuting the bitplanes, setting them to a constant value, and so on. Consequently it is worthwhile to learn how to create such rules.

5.1 Shannon forms

Their simplest description is through four boolean functions, one for each plane, which in turn are functions of four boolean variables, likewise the values of the cells in each plane. Either of the Shannon canonical forms would give a concise description of such a function; for example

$$f(x_0, x_1, x_2, x_3) = \sum_{0,1} f(i_0, i_1, i_2, i_3) x_0^{i_0} x_1^{i_1} x_2^{i_2} x_3^{i_3},$$

wherein the sum represents boolean OR, the boolean AND is implicit, and superscripts differentiate between complementation (0) or not (1).

Some functions are considerably simpler than their canonical form, which is nevertheless a good representation to use.

To describe the canonical form, each of the sixteen coefficients must be specified for each plane; strictly speaking, each one should be binary, but hexadecimal coefficients could be used for economy of representation. The coefficients of all four planes could be combined to give a single sequence of sixteen hexadecimal numbers; alternatively, each individual plane could get a four digit hexadecimal number, which would be more compatible with defining separate rules for each of the bitplanes.

The function `inhx(f)`, whose argument `int f[4][16]` is composed of the four coefficient vectors, installs a $(16,0)$ automaton in the CAM (with a zero alternative), assisted by the table generating function `shancf` which has some additional arguments which are not visible to the user of `inhx`.

5.2 Permutations

The simplest, but one of the most practically useful mappings of the bitplanes, is to permute them; in slightly greater generality some particular plane, its complement, or a constant value might be assigned to each plane. The CAMEX function `apzoc("pqrs")` serves this purpose. Each of its four arguments can take one of the values

a - plane 0
b - plane 1
c - plane 2
d - plane 3
A - complement of plane 0
B - complement of plane 1
C - complement of plane 2
D - complement of plane 3
u - constant value 1
z - constant value 0,

assuming the bitplanes to follow the same order as the arguments. Thus `apzoc("badc")` would imply the exchange of the even planes with the odd planes, `apzoc("cdab")` the exchange of CAM-A with CAM-B, `apzoc("Dabc")` a long cycle in which plane 3 is complemented as it is copied into plane 0.

The simpler mnemonics afforded by these letters is why `apzoc` has a character string

argument rather than using plane numbers directly. Otherwise it is hard to find a nice symbol for the constants, or for the complements.

It is just as easy to create copies or to fill planes with `apzoc`; `apzoc("abaz")` would place a copy of plane 0 in plane 2, while clearing plane 3 and leaving the other two untouched. Clearly, permitting CAM boards to perform all the work is far preferable to extracting the contents of the bitplanes, then modifying and replacing them.

5.3 Boolean combinations

Slightly more complicated than copying or exchanging planes is adding what is in one plane to another, or extracting a portion of one plane and placing it in another. Symmetric boolean functions, such as OR, AND, or XOR between planes will do the work required; since the result always replaces one of the arguments, only the second plane has to be specified in each case.

The result is three more functions in addition to `apzoc`, namely `orple`, `anple`, and `xorple`, all of which have the same style of argument. Thus `orple("aacd")` would move all the live bits in plane 0 over to plane 1 while preserving the live bits that plane 1 already had; the other planes are preserved by self ORing, including plane 0 itself.

If the copying were repented before either plane was modified further, `xorple("zazz")` would erase all the bits which were moved over, including those which were already there; of course, one cannot invert an OR completely. This time `z`'s are needed to preserve the remaining planes.

5.4 Counters

Since the center of each plane is available to all the others, it is sometimes useful to construct a counter from the spare cells in the remaining planes, with intent to incorporate it in the evolutionary rule. Automata modelling the Zhabotinsky reaction frequently use this technique to prolong the refractive interval of cells in the base plane.

If the entire CAM is dedicated to counting, it becomes an assemblage of four bit counters which can span the range from 0 to 15, something which can be accomplished in many ways. The most obvious is to take the bits in the order of the planes, incrementing the four bit hexadecimal digit derived thereby in the accustomed fashion. Apart from permuting the bits, there are other sequences such as the Gray code which are sometimes useful. Any of these counters can be installed using `inhx` with a suitable argument.

If a counter is to be combined with the evolution of a Moore automaton in plane 0, only three bits remain for a counter, restricting its range to 8. A similar range is available to a (2,1) von Neumann automaton in plane 0, but only a two bit counter with a range of 4 is available to a (4,1) von Neumann automaton occupying planes 0 and 1.

The subroutine `inzh()` installs the Moore neighborhood, 3-bit Zhabotinsky rule, calling `zhatobin(t,i,j,p2,p3)` in the process. This latter subroutine uses the global variable `zhtbl[a0][g][n]` to construct a rule table `int *t` suitable for use in plane `j+2*i)` when the center cells in the opposite CAM are `p2` and `p3`.

Meanwhile, `a0` is the cell in plane 0, which can be either infectious ($a0 = 1$) or not ($a0 = 1$), in generation `g` when surrounded by `n` infectious neighbors; editing `zhtbl` is an option accessible through the menu `f1`.

5.5 Half-CAMs

While it is rare that the housekeeping which can be performed under the guise of a $(16, 0)$ automaton ever reaches the full generality of the Shannon canonical form, neither does it always have the permutational simplicity represented by `apzoc`; boolean combinations of pairs of planes, especially within CAM-A or CAM-B are especially common. The trace option, in which plane 1 records all the cells in plane 0 which were ever active, is a good example of the use of a plane OR.

Because the table layout of the CAM favors half-CAMs, and because they involve fewer variables, it is feasible to define functions such as `booltab(r,`f_{11}`,`f_{10}`,`f_{01}`,`f_{00}`)`, whose arguments f_{ij} are the coefficients in the two-variable Shannon canonical form:

$$f(x_0, x_1) \quad = \quad \sum_{0,1} f(i_0, i_1) x_0^{i_0} x_1^{i_1},$$

There are sixteen such functions $f(x, y)$ altogether; their common names (using capital letters for complements) are:

0	0000	and	1000
nor	0001	xnor	1001
Xy	0010	y	1010
X	0011	X+y	1011
xY	0100	x	1100
Y	0101	x+Y	1101
xor	0110	or	1110
nand	0111	1	1111

If there were somehow uniformly short names or abbreviations for all these functions, especially single character abbreviations, mnemonic rather than numerical arguments could have been used; as it is this table or something similar will probably enjoy frequent consultation by someone setting up the functions.

5.6 Video display

Although it is not an automaton, the selection of a color map involves the creation of four boolean functions of four variables to obtain the signals driving RED, GREEN, BLUE, and INTENSITY. The Shannon canonical form gives a direct representation wherein the coefficients of a 4×16 matrix describe the signal required by each combination of values for the variables.

Psychologically, the job is slightly harder because of negative logic; according to the CAM hardware, a boolean TRUE suppresses the color rather than enabling it, making the other Shannon form,

$$f(x_0, x_1) \quad = \quad \prod_{0,1} \{ f(i_0, i_1) + \sum_j x_j^{i_j} \},$$

slightly more appropriate. Only those terms for which $f(i_0, i_1) \neq 1$ need be included in the product when writing the expression on paper, but machine representations prefer to deal uniformly with the full vector of coefficients.

5.7 Margolus neighborhoods

Another context for the Shannon canonical form lies in the definition of rules of evolution for the Magolus neighborhoods; eight binary variables now enter the picture instead of two or four, increasing the number of coefficients that have to be defined.

In turn, many such functions are required, according to the existence of two planes, four coordinate parities, and the alternation of rules between generations; their principle of formation is still the same.

With so many coefficients, it is hardly surprising that procedures would be developed for defining classes of Margolus automata depending on relatively few parameters.

6 The (4,1) automata

The CAM board supports three kinds of neighborhoods — the von Neumann neighborhood in which only lateral neighbors are considered but from two planes, the Moore neighborhood with both lateral and diagonal neighbors but from a single plane, and the Margolis neighborhood. Consequently (4,1) automata are of the von Neumann type; each of the two plane pairs can have its own automaton if that is desired.

In fact, each of the four planes can even have its own (2,1) von Neumann automaton, if the evolution table for the odd planes ignores the neighbors in the even planes, and conversely. Although this represents something of a waste of table space, it is a good way to work with the simpler automata, albeit within a more ambitious hardware environment.

6.1 Shifting planes

Just as the zero-radius automata can be very useful for housekeeping purposes without being of particular intrinsic interest, the simpler von Neumann automata can also be exploited to arrange the bitplanes. Shifting, necessarily confined to the four principal directions, but applied independently to each bitplane, is a good example.

The function `vnshift(t,l,m)` creates a table `int *t` for shifting one plane; it is just as easy to copy the central cell as to complement it, according to the parameter `int l`. One of the planes of the pair must be ignored; the parameter `int m` tells which.

In the four state von Neumann "plane," the states can undergo one of the 24 permutations of a set of four objects, greatly enlarging the concept of a complement. In terms of binary planes, this amounts to passing bits from one plane to the other as well as from one neighbor to another.

Since few housekeeping chores are that elaborate, none of these rules has yet been given a special function of its own within CAMEX. For simple shifts it matters little whether the plane is quaternary or binary.

6.2 Shearing planes

Typically, one CAM board consists of two halves, within each of which there is an evolution plane and a function plane for Moore neighborhoods, and either one single evolution plane-pair or else two independent evolution planes, for von Neumann neighborhoods. Inasmuch as the central cells of all the planes are available to the CAM update mechanism, some of the planes can exert a binary influence upon the choice of rules in the remainder.

If the selection plane is static, this means that some cells evolve by one rule set, the remainder by a second set. The selection plane itself could evolve, resulting in a dynamic process. The extreme case would be a formally defined composite automaton, but oftentimes very simple relationships suffice to get quite practical results.

6.3 Rotation by shearing

Consider shifting part of one plane according to specifications contained in another; to shear a plane, lines should be shifted sideways, some more than others. Thus cells in the sheared plane should first shift along a line while the control line is being copied alongside itself.

On the next pass, two lines will shift while a third line is being added to the control region. The total shift will be equal to the depth of each line in the control region, the

base line having shifted a distance equal to its full depth, the line at the top will have shifted just one cell, while the remainder have yet to move.

Planes are not commonly sheared as a visual exercise, but it turns out that the composite of three shears produces a rotation; consequently images can be rotated within the CAM memory without outside intervention, via the evolution of appropriate automata. To see how this can happen, consider the matrix representation of a shear parallel to the x-axis:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

Equivalently, a y-shear has the form

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ b & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

Compounding the shears requires multiplying the coefficient matrices; a product of three shears gives

$$\begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ b & 1 \end{bmatrix} \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1+ab & a(2+ab) \\ b & 1+ab \end{bmatrix}.$$

If $a = 1$, implying a unit shear to the left, and $b = -1$, implying a unit shear downwards, the final coefficient matrix is

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix},$$

which is readily recognized as a clockwise rotation by 90^0; this relation persists even with the CAM board's cyclic boundaries. Reversing the signs of a and b reverses the sense of the rotation; the center of the rotation is the point of intersection of the original guide lines.

Rotation is not an instantaneous process; the procedure just described requires three sweeps through the whole screen. The process ought to be compared to reading the contents of one or more bitplanes into an external memory, then restoring them, having modified their contents separately or else in conjunction with the restoration. The accessibility of the external memory has to balanced against the restricted range of any internal modification.

6.4 Cross shifting

If shifts and shears can be combined to produce rotations, maybe they could produce reflections as well; but this is not possible. Shear matrices have determinant +1, but the determinant of a reflection is -1, and so cannot have arisen from a product of shears.

However, cells can be moved back and forth between two planes while reversing their linear sequence, which is the equivalent of a reflection. Suppose that plane 0 is shifting north while plane 1 is shifting south. Along the x-axis (or any other east-west line), swap the cells from one plane to the other. Northbound cells turn over and head south, and conversely; after 256 steps each vertical column has turned around completely and the reflection has been performed. Strictly, the full planes should be swapped one last time to bring each cell back to its original plane.

Vertical reflection is the consequence of splitting vertical counterflows along a horizontal line; by symmetry, splitting horizontal flows along a vertical line should result in a horizontal reflection. In both cases the mirror coincides with the control line. Even further, reflection in a diagonal can be accomplished by combining crossflows rather than counterflows, split this time along diagonal lines.

6.5 Symmetries of a square

Since the CAM bitplanes have as many rows as columns, the symmetry operations of a square can be performed upon them, consisting of both rotations and reflections. To do so requires rule tables corresponding to shifts, shears, and cross shifts; of these the shifts (whose table is generated by `vnshift`) have already been described because they are also responsible for translations.

Shears require a control plane containing a sweeping curtain, whose rule table is generated by `vscurtab(t,l,p)`. The argument `int *t` is the array which will contain the table, `char l` specifies the direction of expansion of the curtain; `int p` identifies which plane is destined to hold the curtain, but only its parity matters.

The function to generate the table for a cross shift is `vncshift(t,l,m)` with the same arguments as `vnshift`. Planes 0 and 1 are either shifted or cross shifted, according to the central cell taken from plane 2; plane 3 remains unaltered.

The subroutine `inshth(c)` is designed to set up the rules which will execute all the shifts required to generate a given reflection. The argument `*char c` is a character string `pqrs` specifying the normal shift in planes 0 and 1 plus the cross shifts to be performed when directed by the control plane. The letters range over the same values found in the function `apzoc`; by using complements additional effects can be achieved.

Values of `c` leading to the reflective symmetries of the square are shown in the following table.

reflective symmetry	c	line
horizontal ref	nssn	y=0
vertical ref	weew	x=0
diagonal ref	enne	x=y
antidiagonal ref	esse	x=-y

6.6 General (4,1) automata

Although numerous specialized effects result from mixing all the planes on a CAM board, studying the simplest classes of automata remains one of the major applications of the board; nor should it be forgotten that knowledge of the elementary constituents facilitates the analysis of composite automata.

With its present layout, the most complicated elementary automata that can reside within a half-CAM are those with a (2,1) Moore neighborhood, requiring a table with 512 binary entries, and those with the (4,1) von Neumann neighborhood, requiring 1024 quaternary entries. Of course, a (3,1) von Neumann automaton with a 243 entry trinary table could be accommodated as a special case of the (4,1) automaton, but no space would be saved thereby, either in the bitplanes or in the tables.

The size of their tables makes it difficult to deal with general automata, leading to two alternatives. Many simplifications (such as totalistic rules) have fewer parameters, allowing the full table to be generated without the user's active participation. The other is to look for forms of presentation which will offer increased understanding of the process of editing the rule set, or at least make it more systematic.

Maybe there is a third alternative, which is simply to accept a large table as a legitimate object and to learn how to deal with it. As required, facilities can be added to CAMEX to save rule tables on disk, as well as to recover them.

One way for a table to arise is from a CAMEX editing session, wherein a rule is built up neighborhood by neighborhood while experimenting with trial evolutions. But the table could also be constructed by another program, or copied from somewhere.

7 The (4,1/2) automata

One of the specialties of the CAM boards is the Margolus neighborhood, often used for lattice gas simulations. However, with adroit programming and using large pixels, these neighborhoods can be exploited to program the evolution of a $(4, 1/2)$ automaton. By such a designation we mean an automaton with four states using 2×2 Moore neighborhoods, for a radius of $1/2$. The large pixels are not such a disadvantage, given that the evolved state can be centered so that it overlaps each of the four neighbors; the presentation is more attractive than it would be if the new cell had to coincide with just one of its ancestors.

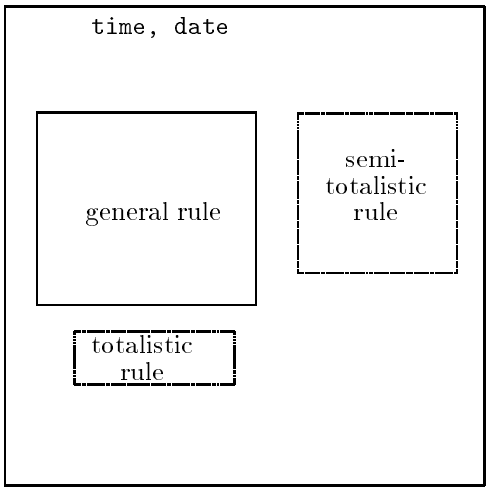

Figure 3: CAMEX 2-D (4,1/2) menu

7.1 Screen display

The screen is laid out so that the full rule of the $(4, 1/2)$ automaton is displayed, the corresponding totalistic rule, and the semitotalistic rule. This latter is obtained by summing the diagonal cells to get one coordinate, the antidiagonal cells to get the other. There is little sense to separating a "central" cell from the remainder with this kind of geometry. If this is all done in text mode, enough space remains to display the traditional help panel on the right hand side of the screen.

There is enough space in the data segment of the INTEL 8086 to accommodate a very rudimentary de Bruijn diagram calculation for $(4, 1/2)$ automata along with the data pertaining to the main program and to the $(4, 1/2)$ option itself. The strips of width 3 which can be computed barely suffice to establish the existence of de Bruijn diagrams, but there wouldn't be enough space for four cell strips even if a separate submenu were allocated for the purpose.

Nevertheless the de Bruijn option occupies a separate graphics mode screen, whose characteristics and operation follow the description given in the section on de Bruijn diagrams for $(2, 1)$ Moore automata.

7.2 Option menu

As usual, the general list of options can be grouped into categories; they are similar for all the different varieties of automata.

7.2.1 The full menu

Alphabetically, the options are the following:

0,1,2,3 - insert state
SPACE, MOR - cursor forward
BACK, MOL - cursor back
MOU - cursor up
MOD - cursor down
INSERT - deposit rule in CAM
DELETE - clear all planes
< - cursor to left margin
> - cursor to right margin
a - generate upper left shift
b - generate upper right shift

c - generate lower left shift
d - generate lower right shift
A - generate upper left eater
B - generate upper right eater
C - generate lower left eater
D - generate lower right eater
e - generate majority rule
j - de Bruijn submenu
m - edit semitotalistic rule
p## - product rule
R - run (even number of steps)
s - single step
S - continuing evolution
t - edit totalistic rule
u - reversible rule
v - reversed reversible rule
w# - number of a (2,1/2) rule
y - generate random ellipse
Y - generate random plane pair
z - clear the rule table

The items for which an argument is marked by a # require a number to be typed from the keyboard; it is not echoed and if one feels that a mistake has been made it should be retyped.

Initially only the general table is accessible, and is the only one which is shown. The options `t` (for totalistic) and `m` (for semitotalistic) manifest themselves by showing their own specialized little menu. However, to avoid constant switching between menus, the general menu is always updated whenever one of the specialized menus is changed; for the same reason all of the commands for which it makes reasonable sense to do so are repeated in each menu.

7.2.2 Defining the table

Each of the tables has a cursor, consisting in illuminating its table entry with a distinct color; to determine the value of the table at that position, it is only necessary to type the number of the state. The cursor will advance automatically, except that it will not change rows when there are more than one.

The cursor can be positioned without altering any states by using the arrows; space and backspace can also be used. If a mouse is available and programmed to emit arrows, that is another way to position the cursor.

Creating a table on the screen sets up a data table in the program's data space, but does not place it in the CAM. The tables persist throughout a CAMEX session; changing demonstrations always leaves the tables of previous demonstrations intact, so that it is always possible to return and continue without interruption.

However, the bitplanes and the tables may have changed during the interim; INSERT will install the table at any time, either to continue a previous experiment, or to set up the table which has just been edited.

Another special case, `e`, is one in which a cell joins its neighbors whenever all three of them agree. But the agreement, if any, is transitory and keeps shifting.

7.2.3 Arranging bitplanes

Bitplanes are never saved, nor are they automatically cleared, so their state is cumulative from one operation to the next; an advantage is that data may be passed from one demonstration to another. Nearly every demonstration, as well as the main menu, allows the bitplanes to be cleared; also to generate one or more varieties of random planes. Insofar as possible, the keys DELETE, `y`, and `Y` are reserved for this application.

7.2.4 Special rules

Several of the options generate especially common or interesting rules. For example, `a, b, c, d` generate shifts along the four di-

agonal directions; the new cell simply copies one of its neighbors.

An "eater" type of rule is one in which a cell acquires the state of whichever of its neighbors is poised to "eat" it; the food chain is generally cyclic, so that 1 eats 0, 2 eats 1, and so on until 0 eats 3. Lacking a central cell, there are four candidates as to who will be the meal, consequently four rules, `A`, `B`, `C`, `D`.

7.2.5 General rules

Several general classes of rules, which might otherwise all be treated separately, are included as special cases of the $(4, 1/2)$ rule; these are totalistic rules whose submenu is invoked by `t`, the even-odd rules (a variant of semitotalism) accessible via `m`, and the reversible rules. The former have been endowed with all the options of the general menu which make sense (shifts are inappropriate, for example), so that trials can be performed without shifting menus.

7.2.6 Reversible rules

The Fredkin style rules contained in the menu require the Wolfram number of a $(2, 1/2)$ rule, a single number in the range 0, 65,535. It is received, without echo and without corrections, by the option `w`, to be displayed in the upper right hand corner of the screen. If it was entered incorrectly, it is easy enough to try again.

The options `u` and `v` generate one or the other of a pair of mutually reversible rules, but they only create a rule table in the computer's memory; INSERT is needed to transfer the rule to the CAM.

The option `R` should be used instead of `s` or `S` to run the evolution; it guarantees the even number of generations which must be respected on account of the parity of the Margolus neighborhood. Such being the case, the undoing of an evolution can be observed easily.

7.2.7 Product rules

Two Wolfram numbers are needed to generate a product rule; they should be supplied immediately following the option `p`. All numbers have to be terminated by typing any non-digit; errors can only be corrected by retyping everything once again. After the table is formed, INSERT gives it to the CAM.

7.2.8 Evolution

The letters `s` and `S` are reserved throughout CAMEX to start the CAM; the former for one generation, the latter to run until interrupted.

To use the Margolus neighborhood and to accommodate the lack of a central cell, big pixels are used for the $(4, 1/2)$ automata. However, random bitplanes are not created with big pixels, which ordinarily would not matter because the initial step ignores three of the four pixels in a cell. Thereafter the cells are all uniform, and there is no further problem.

To get a reversible rule to function correctly, it is essential for the pixel to be uniform and in phase with the Margolus neighborhood; this is most easily assured by running through an even number of generations before installing the new rule, and is the reason for the option `R`.

7.2.9 The de Bruijn diagram

The de Bruijn diagram, and the use of the de Bruijn option, are both discussed in a general section devoted to that topic.

7.3 The automaton itself

The reason for working with automata of radius 1/2 is their simplicity; one has to maintain a balance between such small automata they are trivial, or so large they are computationally impossible. Having four states means that the widest de Bruijn strip is only three cells wide; yet automata of radius 1, the next larger size, will not let even this small diagram fit the available memory.

7.3.1 Cartesian product

In turn, four is the smallest number of states allowing the exploration of a non-trivial state space, namely the cartesian product of two binary state spaces. Aside from the classical cartesian product, in which the two factors evolve independently, other combinations are possible, including Fredkin's construction of reversible automata.

Suppose that there is some one dimensional $(2,1)$ automaton A whose evolution follows the function $\alpha(x,y)$, and a second, B, whose rule is $\beta(x,y)$. Then their cartesian product $A \times B$ is another $(2,1)$ automaton whose states are cartesian products of the states of A with those of B, having the rule of evolution

$$\Phi\left((a,b),(c,d)\right) \quad = \quad (\alpha(a,c),\beta(b,d))\,.$$

To provide a similar definition for two-dimensional automata requires no more than extending each function to four variables rather than two, the same as needs to be done for any other dimension or neighborhood structure.

Every pair of binary rules leads to a distinct quaternary automaton, but even so, the number of quaternary rules which are cartesian products is a very small fraction of the total number of rules.

7.3.2 Reversible automata

Another construction in the cartesian product of state spaces yields automata whose evolution can be reversed; it is based on ideas originating with Edward Fredkin. This time we need just one rule, $\varphi(x,y)$, with which the following composite rule of evolution can be created:

$$\Phi\left((a,b),(c,d)\right) \quad = \quad (b \wedge \varphi(a,c),a)\,.$$

This definition has a leftward orientation, with an entirely similar definition leaning to the right. Likewise, the exclusive or could be replaced by any other invertible function; for binary automata the only alternative would be the exclusive nor.

To undo the evolution, consider the evolution of three successive cells:

$$\begin{array}{ccc} (a,b) & (c,d) & (e,f) \\ (b \wedge \varphi(a,c),a) & (d \wedge \varphi(c,e),c); & \end{array}$$

we would like to recover the state (a,b) from the second line. Although a is readily available, it is necessary to liberate b from the combination $b \wedge \varphi(a,c)$. Given that both a and c are available, $\varphi(a,c)$ can be calculated, then used to release b from the invertible combination in which it is bound.

Altogether this is a process embodied in the rule

$$\Psi\left((a,b),(c,d)\right) \quad = \quad (b,a \wedge \varphi(b,d))\,,$$

which must be the rule which will reverse the evolution resulting from the application of the first.

To apply these ideas in two dimensions, consider the following layout:

$$\begin{array}{ccc} (a,b) & (c,d) & (e,f) \\ (j \wedge \varphi(a,c,g,i),a) & & (l \wedge \varphi(c,e,i,k),c) \\ (g,h) & (i,j) & (k,l) \\ (p \wedge \varphi(g,i,m,o),g) & & (r \wedge \varphi(i,k,o,q),i) \\ (m,n) & (o,p) & (q,r) \end{array}$$

wherein the rule of evolution is evidently

$$\Phi\left((a,b),(c,d),(e,f),(g,h)\right) = (h \wedge \varphi(a,c,e,g),a)\,.$$

As before, there is enough information in the second generation cells surrounding the first generation cell (i,j) to recover (a,b); the appropriate rule is:

$$\Psi\left((a,b),(c,d),(e,f),(g,h)\right) = (h,a \wedge \varphi(b,d,f,h))\,.$$

Again following the one dimensional case, these definitions can be oriented toward different directions, just as the connective $\wedge$ can be assigned different meanings. Such variants do not necessarily yield all possible invertible rules; historically very few were known until Fredkin proposed "second order" rules, of which these are a variant.

Every binary rule leads to a distinct pair of reversible quaternary rules (some of which may be self-reversible, and some pairs of which may have been generated by a pair of binary rules), even when the binary rule itself is not reversible; there are also cases in which the binary rule was already reversible.

7.3.3 Universal emulation

Given enough states, and a willingness to interpret the evolution — say by sampling every second generation — any cellular automaton can be emulated by a radius 1/2 automaton of the same dimension.

Unfortunately four states, the number available in a half-CAM, is insufficient; emulation experiments cannot be performed with equipment having its present configuration.

Had it been possible, emulation of a Moore automaton would have been accomplished along the following lines. Any large neighborhood can be decomposed into smaller tiles, not forgetting to make allowance for overlap. Arriving at 2×2 tiles, the size of neighborhoods in a radius 1/2 automaton, and stopping with individual cells, we find a large hierarchy of subtitles.

The entire collection of subtitles is to become the state set of the emulating automaton, which still requires a rule of evolution. That rule consists in promoting small tiles into larger tiles until the size of the neighborhood of the automaton to be emulated is reached. At that point, the neighborhood evolves into a single state of a single cell according to the original rule; then the process commences anew.

The scheme is simple enough, yet the number of intermediate states is quite large; in one dimension it requires six states for a (2,1) automaton — two are the original binary states, four more are pairs. Pairs of pairs overlap to form the original three cell neighborhood, whose image can be incorporated into the rule immediately.

A second rule, to accomplish the emulation, requires that a configuration of singlets turns into a configuration of pairs, which evolves back into singlets. To see pure singlets, only odd generations ought to be consulted, but they will evolve into each other following the first rule.

The evolution of mixed configurations, being irrelevant to the emulation, may be defined according to convenience.

8 The (2,1) automata

When Moore neighborhoods are used in a CAM, there is only space enough in the tables for a single bitplane, together with the centers of the three remaining planes. However, there is an additional constraint, namely that the tables always refer to just the even bitplanes; the result has even planes evolving, but odd planes displaying nothing more than functional information about their partners. Moreover, the arrangement is confined to binary automata.

8.1 Generalities

Echoes and traces are two common uses of the odd bitplanes, although echoes may be made to *any* of the other bitplanes. Indeed, Zhabotinsky type rules incorporate the echo into counters and thence into the rule of evolution itself.

Simple echoes have artistic value, and are even useful as indicators of the ages of cells. Cumulative echoes are often employed in automata such as *Life*, to give an indication of how far an initial configuration expanded before it stabilized or died out.

But *Life* is also a good example of an automaton in which a trace may be used to advantage. As is well known, gliders have a special significance in *Life*, so one might ask about where the gliders are. Fortunately, a glider fits quite snugly into a single Moore neighborhood; a good rule for the odd plane accompanying a *Life* plane would be to accumulate those cells whose neighborhoods ever contained a glider. Gliders have two phases, move in four directions, and have mirror images; the rule can detect any or all of them.

The result of such a composite rule looks somewhat like a nuclear emulsion. Many neighborhoods will contain gliders, surrounded by live cells which interfere with the glider; there ought to be quite a few marked cells. Indeed, rough estimates of how many of the $2^9 = 512$ neighborhoods would be shaped like a glider show that they are not at all uncommon, perhaps comprising 3% of all neighborhoods. But whenever a glider breaks loose or travels freely, it will leave a diagonal line behind; those are the tracks to be counted.

8.2 Moore (2,1) automata

The binary 3×3 Moore neighborhood in a plane contains nine cells, meaning that there are 2^9 or 512 different neighborhoods, yielding an incredible 2^{256} different automata. After working with cellular automata for a while one becomes accustomed to such large numbers, as well as to techniques to reduce them to more manageable proportions. The vast majority of automata are passed over in the process; those which remain can be studied more systematically.

8.2.1 Totalistic rules

By far the most symmetric restriction of a rule table is to consider only the *number* of neighbors of a cell, but not their spatial distribution. The result is called a *totalistic* rule, because of its dependence on the sum. For the $(2,1)$ Moore neighborhood, there are nine neighbors, therefore ten different sums. Sums of 0 and 9 can occur in only one way, but there are nine neighborhoods containing but one cell, and nine more with eight neighbors. In general, the binomial coefficients $9!/k!(9-k)!$ tell how many neighborhoods contain k cells.

A menu display for defining a totalistic rule will typically display the various sums along a row, the value which the cell will acquire in the new generation being shown in the

matching position in the row below. The position of a movable cursor shows where new values may be inserted, thereby changing the rule.

8.2.2 Semitotalistic rules

A slightly more versatile way to define a rule table is to separate the central cell from its neighbors, treating their sum according to the state of the central cell. *Life* is a rule of this kind, because live cells evolve differently from the others, but in both cases the decision depends on how many of the remaining eight neighbors are present. The adjective describing these rules is *semitotalistic*.

The display for a semitotalistic rule would consist of two rows, one for birth and one for survival, with a common header showing totals. The number of neighborhoods with a common total and central cell would be $8!/k!(8-k)!$, for Moore neighborhoods.

8.2.3 Even-odd-center rules

Most lattices can be partitioned into even and odd sublattices, much as the squares on a checkerboard can be colored red and black, for example. Sums can be taken separately in the two lattices, since the neighborhoods will reflect the same partitioning; but when there is a central cell, it can be considered separately, to preserve the balance between the two classes. Rules reflecting this partitioning are called *even-odd-center* rules.

Considering the neighborhood in isolation from the lattice, the concept of even and odd can be interpreted fairly liberally; for instance as the top of the neighborhood and the bottom of the neighborhood. Sometimes interesting variants on the rules result from such creative partitioning.

Displaying an even-odd-center rule requires two 5×5 matrices for a Moore neighborhood, because there are two classes with an occupancy running between zero and four, modified by the two values of the central cell.

8.2.4 Symmetric rules

Whenever the lattice underlying a cellular automaton is symmetric, there are plausible reasons to work with rules which have the same symmetry; $(2,1)$ Moore neighborhoods have square symmetry on account of the lattice, with additional symmetries due to complementation which are sometimes taken into account.

Although there are eight symmetry operations for a square, not all symmetry classes have eight members; for example there is only one member of the class containing the zero neighborhood. The symmetries preserve distance, so that the central cell always stands by itself; nor are diagonal neighbors ever mixed with lateral neighbors. All told, there are 102 symmetry classes, 51 each for the two states of the center cell.

It would be an option to leave out reflections, to consider only rotational symmetries of the neighborhoods. This might split some of the symmetry classes, but hardly all of them; choosing this degree of specialization assumes that the handedness of the rules have importance.

8.2.5 General rules

Sometimes it is necessary to study the most general rule possible, given the neighborhood and state set, making it necessary to present the full rule for editing; the most compact way to do this is to show two 16×16 matrices, again differentiating them by the central cell to which they refer.

The circumstances under which a really general rule has to be defined are often that a given course of evolution is being sought.

So the editor for such a rule ought to incorporate provisions for trial evolutions; also for marking portions of the rule which have already been settled upon to distinguish them from those yet to be decided. Finally, the rule ought to be saved somehow, if tedious manual transcription, or laborious repetition at a later date, is to be avoided.

8.3 Hexagonal (2,1) automata

There are certain sublattices of the general square lattice which can be provided for with a special editor of their own, to conceal the full generality which would only distract the user. The hexagonal lattice is one of them, the square lattice of radius 1/2 is another.

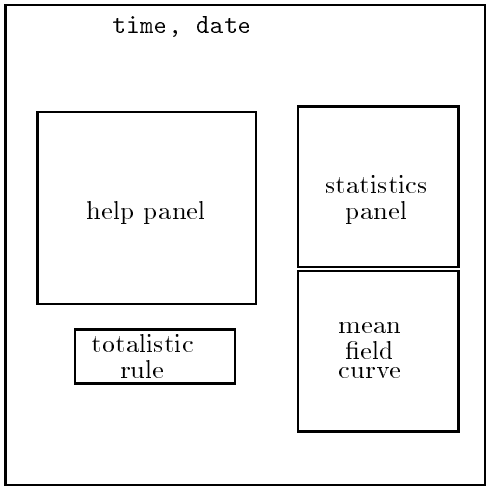

Figure 4: Hexagonal (2,1) menu

As with automata operating with full neighborhoods, it is convenient to introduce the degrees of totalistic, semitotalistic, symmetric, and so on.

The menus for all these options are very similar, as are the screen layouts, as illustrated in Figure 4.

The options are typical, namely:

- 0, 1 define state
MOA - toggle state
SPACE, MOR - advance cursor
BACK, MOL - cursor backward
INSERT - load rule into CAM
ERASE - clear CAM bitplanes
s - single step of evolution
S - generations of evolution
t - graph mean-field probability
u - graph Bernstein monomial
x - choose a random rule
y - random ellipse
Y - random full plane
z - clear graph area
! - function key menu
? - show keyboard menu

The probability panel shows the percent of live cells in each generation, supposing that a compatible color scheme has been chosen for the CAM; if the results appear unreasonable, try adjusting the colors to activate the event counter correctly.

The graph panel shows the mean field theory curve for the rule; since the diagonal is also displayed, the fixed points and their stability can be determined visually, to be compared with the density bar chart in the probability panel.

8.4 Moore (2,1/2) automata

By working with a corner of the radius 1 Moore automaton, radius 1/2 may be accommodated. There are relatively few of these automata and their properties are not very complicated; nevertheless it is useful to have access to them for comparison and for completeness.

Defining specialized rules is hardly worth the trouble, but is done to a slight extent for purposes of comparison.

9 One dimension

Although CAM boards are primarily designed for two-dimensional automata, one-dimensional rules are easily established; it is only required that all but the one dimensional neighbors be ignored. Discarding vertical neighbors makes each row of the display a one-dimensional automaton; if columns were preferred, the horizontal neighbors should have been discarded.

9.1 Generalities

An unimproved display tends to be confusing, the screen being composed of a large number of automata functioning in parallel. Some slight preparation will create the illusion of successive generations of evolution.

First, a single line has to be copied throughout the screen, perhaps by loading an evolutionary rule which will create the copy. Simply loading one single line throughout the bitplane, which can be done by `svuline(c,p)`, would probably be faster.

Next, use a sweeping curtain to age successive lines, leaving the remainder untouched. Once the full screen has been swept out, uniform evolution will produce a moving display in which successive lines are a single generation apart.

One-dimensional $(4,1)$ automata result from using von Neumann neighborhoods in CAM-A, but $(3,1)$ or even $(2,1)$ automata would be the result of introducing rules which ignored some of the states.

Whereas only $(2,1)$ automata can be based on Moore neighborhoods, the possibility of taking neighbors from an adjacent line allows a static display with a moving line of evolution, rather than the display moving away from a static line which is the only possibility with von Neumann neighborhoods.

9.2 Programming niceties

The subroutine `inod(w)`, whose argument `w` is the Wolfram rule number of a one dimensional $(2,1)$ automaton, will install the corresponding table in CAM-A. No other record of the rule is left behind, since it is so easy to regenerate from the rule number.

Installation of one-dimensional rules is actually a two step process; the first part is to create the rule table, the second is to create the unified screen image. The two are combined in `inod`.

For a $(3,1)$ automaton, `oned31(r0,r1,t)` transforms `char t[27]` into two tables, `char *r0` for plane 0 and `char *r1` for plane 1. It is usually called by `inod31(r0,r1,t)`, `t` having previously been prepared, say by choice of the option offered by `f1`.

Members of the pair `oned41(r0,r1,t)` and `inod41(r0,r1,t)` together render the same services for $(4,1)$ automata; this time `char t[64]` would be generated by another `f1` option.

In both cases suitable global variables are provided, `char *tbl31` and `char *tbl41` respectively, so that the rules will always be retained between consultations of the editor, and for the use of other subroutines.

Several other one-dimensional automata can be emulated by a CAM, but the techniques involved do not produce such attractive visual images as the ones mentioned. Although the rules for one dimensional automata make a pleasant adjunct to CAMEX, the analysis is nowhere as detailed as that provided by the program set LCAU; probabilistic surveys, de Bruijn diagrams, and the calculation of ancestors are all missing.

9.3 The collection LCAU

Although there is a certain overlap between the domains of LCAU and CAMEX, it is not extensive. CAMEX is primarily two-dimensional with some interesting extensions to three dimensions, whereas LCAU is entirely one-dimensional. The CAM's speed of operation is essential for interactive studies in two dimensions, but one dimensional automata already operate at nearly the same speed because of the redundancy of the plane image; there would be an advantage to a special connection converting the plane into one long line if someone required such a configuration.

Because the calculations are simpler, one dimensional automata can be explored in considerably more detail than higher dimensional automata. Likewise, the cells can have many more states than is convenient for the CAM hardware. A review of the properties which are calculable shows some of the things which could be added to CAMEX in the course of further development, especially as the speed and memory capacity of computers continues to grow.

9.3.1 Evolution

The one area in which the performance of CAMEX, LCAU, and various other programs is comparable is the graphical display of the evolution of automata. Of course there are differences in speed, surface area, number of cells, and the like; but all programs offer this facility in some form or another.

9.3.2 Probability

The statistical properties of automata, both theoretical and empirical, are of the utmost importance. The CAM event counter, accompanied by suitable programming, can be used to collect different kinds of data; naturally the programming increases in complexity as the quantity of data to be extracted increases.

The probabilistic section is not only a major constituent of all the LCAU programs; they incorporate a fair diversity of analysis. The calculation of a "mean field curve," which describes the change of probability from one generation to the next, assumes that the state probabilities are not correlated among neighboring cells. The assumption has varying degrees of validity from one automaton to another, but it always provides a valuable frame of reference. More elaborate calculations, in terms of "block probabilities," are also included in LCAU.

9.3.3 De Bruijn diagrams

With time, the evolution of finite automata becomes periodic; automata whose lattice is a ring or a torus already fit this description, although the length of the period may be extremely long, as well as the time required to pass through the transients and arrive at the final cycle.

The systematic way to decide upon the periodic behavior of a cellular automaton is to use de Bruijn diagrams, which are simply maps showing the relation of neighborhoods to one another, annotated with details concerning interesting aspects of their evolution. All the LCAU programs contain a submenu devoted to the calculation of de Bruijn diagrams.

For neighborhoods of large radii or automata of numerous states, the quantity of information obtainable (or rather, the range of parameters over which it is available, the quantity actually remaining fairly constant) diminishes. Therefore the coverage is more complete for the smaller automata.

For two dimensions and beyond, the quantity of computation required is truly

formidable.

9.3.4 Ancestors

If evolution is important, the reverse must also be interesting. Substantially the same techniques that are embodied in the de Bruijn diagrams can be used to determine the number of ancestors, and indeed, their actual composition. The computations are more arduous, resulting in just a few of the LCAU programs having an ancestor submenu; the prospect has not even been considered in two dimensions in CAMEX, although the theory is just as applicable.

Two aspects of the calculation of ancestors are often given special attention. One is to determine which configurations have no ancestors at all, the so-called Garden of Eden configurations. The other is to find automata for which evolution is reversible, in the sense that every configuration should have just one ancestor.

9.3.5 Designing rules

Another useful facility, which is easier to provide for one dimensional automata than for others, is the ability to edit the rule table while performing trial evolutions. By a process of definition and adjustment, rules can be built up which conform to a prescribed evolution, such as exhibiting solitons and allowing them to pass each other, to mention an example.

9.4 (2,1) automata

One dimensional binary automata are given rather cursory treatment in CAMEX, in contrast to the fairly elaborate analyses which LCAU21 can preform. They are included, like many others, in the list which can be viewed by using the option `f1` in the man menu; the subsequent use of INSERT initiates the ritual of loading CAM's lookup table and creating a suitable image on the screen, to simulate a space-time evolution diagram.

Once the initialization step has been performed, using the subroutine `inod`, it can be repeated to generate another random configuration following the space-time format. However, any of the many other screen-filling options can be used to create configurations for which each line will evolve according to its own destiny. Placing a checkerboard in plane 0 (using `h1` or `H1`) can often be instructive.

For the moment, neither of the options `r` or `l` responds while the binary automaton has been selected; the only variation possible is to change the rule number, which is a parameter that has to be adjusted from the main menu. Its value can be checked by using option `f7`, or changing parameters via the cursor arrows.

The vertical arrows increment or decrement the rule number; to get a different value quickly, say 150, type `=150`. Rule 22, whose behavior is as diverse as any, is the default value. Other interesting rules are 18, 30, 90, but there are many more.

9.5 (3,1) automata

The layout of the menu for one dimensional (3,1) automata is shown in Figure 5, which is typical of the remaining one dimensional menus. Little more is shown that the rule definition, with an additional panel for the totalistic rule, if that option is chosen.

9.5.1 Options

Either of the two rules can be edited by typing the state which should be situated at the location of the cursor, which can also be moved by using the arrows, the mouse if it is connected and programmed to deliver ar-

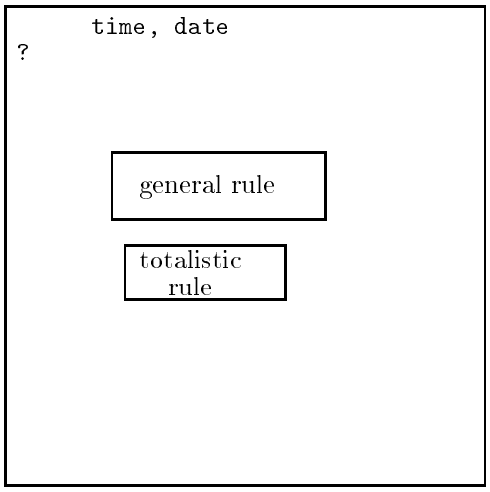

Figure 5: CAMEX 1-D (3,1) menu

rows, or such mundane keys as space and backspace. The new table can be inserted in the CAM's lookup table, evolution set in motion, random tables and screens selected and so on, according to the complete menu belonging to the option.

If they have been provided in a separate file located in the current directory called LCAU31.RUL, demonstration rules may be selected by using the key `f1`, then loaded and executed.

9.5.2 The option menu

The complete collection of options is:

0,1,2 - define state under cursor
TAB - next quad
`f1` - view sample rules
SPACE, MOR - cursor right
BACK, MOL - cursor left
MOU - unmark neighborhood
MOD - mark neighborhood
INSERT - activate rule
DELETE - clear all planes
s - single step of evolution
S - ongoing of evolution
t - edit totalistic rule
x - random protected rule table
X - fully random rule table
y - random planes 0,1

Between automata, the only substantial variation in this menu lies in the range of states admitted, and the appropriate number of neighborhoods or state sums.

9.5.3 Sample rules of evolution

If a file of rules has been given the name LCAU31.RUL and resides in the currently active directory, the function key `f1` will load and display it. In order for the process to work, the file must have been prepared in strict accordance with the specification, that each line contains one rule plus a comment.

The entire line consists of ASCII characters, which means that any editor can be used to prepare it; likewise that rules can be added, removed, or rearranged at will by the user.

The rule itself consists of exactly 27 characters chosen from the set $\{0, 1, 2\}$, followed by a space. The rule will not be shown on the screen, but it will be copied into CHAR *TBL31, which is the appropriate CAMEX rule array, on request.

The comment following the rule will be displayed, and should be chosen carefully to give a good description of the rule. Due to certain inflexibilities of the loading format and display program, it should consist of exactly 32 characters, the first of which is not a blank. There is also a limit of 40 on the number of rules which the file should contain.

9.6 Totalistic (3,1) rules

Once the number of states, or the radius of the neighborhoods, in an automaton becomes

large, it becomes increasingly difficult to describe the rule; the only effective description is a listing of the transition table. Specialized automata are usually easier to describe; one of the most common special cases is that of a totalistic automaton, for which the transition depends only on the *number* of cells in various states, not their specific arrangement.

Accordingly, an option within the general rule is to show the totalistic variant within a submenu; it is convenient to reproduce most of the options within the submenu that were encountered in the principal menu to avoid the continual swapping of menus.

However the `f1` option is not active for totalistic rules, which can nevertheless be included rather easily in the general file when they are desired.

To assist copying down an interesting rule on paper for eventual inclusion in the file, advantage can be taken of the fact that each change in the totalistic rule is immediately reflected in a redefinition of the general rule, which is still shown on the screen.

9.7 (4,1) automata

There is very little difference between the way $(4,1)$ and $(3,1)$ one dimensional automata work in CAMEX, most of those differences being natural consequences of the difference in the number of states and correspondingly, the number of neighborhoods, between the two classes. The description of the trinary programs is equally applicable to the quaternary programs.

Amongst the demonstrations, there are interesting aspects of quaternary automata to be discerned, which cannot be realized with fewer states. For example, the factorization $4 = 2 \times 2$ means that quaternary automata can be the direct product of binary automata. Besides all those automata which are simple direct products, there are reversible automata constructed by applying Fredkin's technique to the direct product.

Other possibilities include the construction of binary counters, for some of which the carry bit acts as a "glider." Other gliders exist, some of which act like solitons. Altogether, the rule set is large enough for quite a bit of experimentation.

Both the $(3,1)$ and $(4,1)$ automata have disk files, called LCAU31.RUL and LCAU41.RUL respectively, which contain a couple dozen or so rules that can be loaded into their rule tables on demand. The files can be altered outside of CAMEX with the help of any text editor, either to remove items or to include additional ones.

The format of the tables is almost self explanatory; the rule is typed using ASCII digits as in any text file, with an identifying comment following. The table in the the file LCAU41.RUL occupies 64 columns whose entries are the digits $0, 1, 2, 3$. The style of the existing file, especially the field lengths, should be followed in making alterations.

Within each submenu, the function key `f1` will search the current directory for the appropriate rule file and display its contents.

9.8 Prospects

So far, a minimal amount of LCAU has been incorporated into CAMEX. In part this has been for reasons of space, which is scarce within the memory model which has been employed; in part is is because the programs already work satisfactorily in their own environment and do not require the CAM in an essential way. It would surely be more productive to concentrate on two-dimensional versions of the features discussed, for the further development of CAMEX.

10 Three dimensions

Some three dimensional automata, namely $(2,1)$ automata using von Neumann neighborhoods, are accessible to CAM boards, given a willingness to work with them plane by plane. Since four planes altogether can be used, plane 0 could be regarded as sandwiched between planes 2 and 3, to create a plane each of whose cells has a complete three dimensional neighborhood of unit radius. Rather than having plane 0 evolve, the new generation could be recorded in plane 1 instead, freeing the CAM board to shift layers internally.

10.1 Logical arrangement

The evolution of a whole stack of planes could be obtained by first loading three consecutive planes from supplementary storage, such as from arrays in the computer's own memory, or files stored on disk. From then on, a composite cycle of evolution would record the evolved plane 0 in plane 1, simultaneously shifting planes 0 to 2 and 3 to 0. Once done, plane 1 would be saved as the new generation, whilst loading plane 3 from the next layer of the old generation, completing the cycle at last.

Backup storage could maintain the two generations as separate, but unless that were necessary for purposes of comparison or something similar; really it is only necessary to save just one single plane to be sure of closing the cycle at the end.

For the moment, allocating a single file of ramdisk memory to each 256x256 plane produces an 8K byte image for each of them. Considerations of the Intel 8086 CPU family, the MS/DOS operating system, and a C language compiler enforce a practical limit of 12 planes. The result is a very "thin" automaton, nevertheless fat enough to obtain useful information; experience will doubtless lead to the necessary programming for more versatile combinations.

Even so, extremely thin automata are interesting as limiting cases, so variants have been programmed whereby automata of 1, 2, 3, or 4 planes can be retained internally and followed at the same rate of normal evolution as any other automaton.

Otherwise evolution requires about ten seconds per plane, with a very flickering performance. This is because evolution occurs in a flash, data movement taking much longer, under a blank screen. Including a programmed delay in the cycle will hold the image longer, at the price of a still further slowdown.

For many automata and for many purposes this style conveys a feeling for the structure and texture of the automaton; some leeway is also provided by the choice of a color map. Evolution can be suspended at the end of each sweep through the planes, allowing static views, or visual planewise inspection. There is a clear opportunity to experiment with alternative visual presentations.

10.2 The module CAMETD

One entire source module, CAMETD, is dedicated to three dimensional $(2,1)$ von Neumann automata; seven binary cells per neighborhood yield 128 different neighborhoods, and accordingly 2^{128} different rules. This great number can be diminished by requiring symmetrically related neighborhoods to undergo the same transitions; and even more by stipulating that the rules be totalistic or semitotalistic. At present only these latter two alternatives are recognized by CAMEX.

The way to work with a three dimensional automaton in CAMEX is to locate one or the

other of these alternatives in the master automaton menu, which can be called up from the main program by the function key `f1`. After making the selection, the option `r` (edit rule) will display a submenu in the graphic mode.

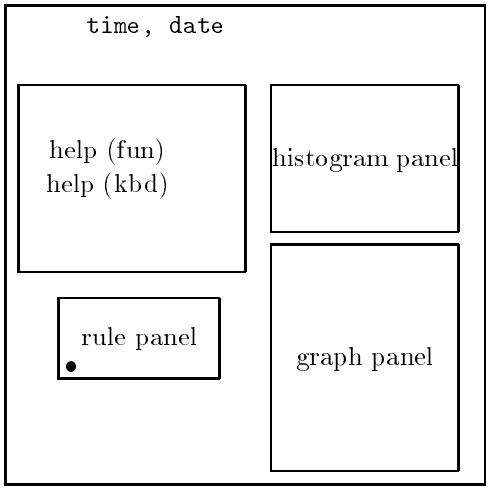

Figure 6: CAMETD console panels

Now the screen will show the usual commentary, together with a panel in which the rule specification may be entered, a graphics panel and a bar chart panel. At the graphics panel the rule's Bernstein polynomial, or mean field curve, can be shown (option `t`); during evolution points of the return map will appear in the same panel.

The bar chart shows the running percentage of live cells, provided that the color map chosen for the CAM/PC only assigns the intensity bit to one plane. If the frequency displays do not seem credible, the color assignment may be at fault.

10.3 Menu options

In alphabetic order, the submenu which currently exists is the following:

0,1 - insert state in rule
C - special color scheme
g - single step "thin" version
G - run "thin" version indefinitely
h - single-plane version
H - random field for h
i - two-plane version
I - random field for i
j - three-plane version
J - random field for j
k - four-plane version
K - random field for k
s - run CAM one step
S - run CAM indefinitely
p - next rule
t - MOX - mean field curve
u - MOB - Bernstein monomial
x - rand rule selection
y - random ellipse in plane 0
Y - put random fields in diskfiles
z - clear graph area
! - show function key menu
? - show the option menu

This menu is not unlike those for any other automaton; but the difference between the pair `S, s` and the pair `G, g` should be noted. Both run the CAM through cycles of evolution, but the cycle is much more complex than normal, and varies according to the depth of the automaton.

10.3.1 Thick

When more than four planes are in use, juggling supplementary memory is required; at present this is done from the RAM disk for a number of planes limited by DOS's buffer allocation. Both the number of disks and the

disk assignment are embedded in the source program; they are set for twelve planes on unit D. Changes require recompilation, which is not a fully satisfactory arrangement.

The regular pair of run commands, S and s, are assigned to this combination; evolution will be slower and blinker than usual. It is necessary to wait out the full sequence of planes in each generation before proceeding to the next generation, but suppressing part of the presentation would not improve the speed given that the display is an inherent part of the cam's operation.

Whatever rule is visible in the rule panel will be placed in the lookup table by INSERT.

All the planes can be randomized to 50% initial density by typing Y, even those in disk files.

10.3.2 Thin

Each of the four shallow depths is given separate treatment with respect to creating the lookup table, initializing the planes, and running the CAM. The rule is still taken from the rule panel, but the connectivity of the planes in the CAM changes with the depth; likewise the preparation of the planes is different.

Thus h loads the table, H the single bitplane for the degenerate case of uniformity through the entire depth. In turn i makes the table, I the two planes, for an automaton whose planes alternate through the depths. A cycle of 3 is governed by j and J, and 4 by k and K.

The special circumstances of evolution have been assigned to the run pair G and g because no swapping of planes is required. Accordingly the evolution will run much more quickly, and also more smoothly because the planes are not shown individually. The merged image is not truly three dimensional.

10.4 Function keys

In turn, the function keys which are active within CAMETD are the following:

f1 - bitplane image subsubmenu
f8 - show next plane
F8 - show first plane
CTL f8 - show plane 0 only
ALT f8 - show previous plane
f9 - show next plane
F9 - primary plane colors
CTL f9 - show plane 0 only
ALT f9 - show previous plane
f10 - next color scheme
CTL f10 - first color scheme
ALT f10 - previous colors
MOA - toggle state
SPACE - MOR - advance cursor
BAK - MOL - move cursor backward
INSERT - activate rule
DELETE - clear all planes

The keys governing the color selection have been included because the color combination affects the quality of the image perception.

10.5 Experiments

A typical three dimensional automaton experiment, after having arrived at the submenu, would be to enter the rule number using arrows, space, backspace, and the numbers 0 and 1 to designate states. For purposes of initialization, a random number generator always provides one, but it may not be interesting. Once set, it remains until reset, even though one leaves the submenu and later returns.

If a two button mouse is available, it should be set to generate arrows for its movements, designated mou, mol, mor, mod in the menu above. In addition, the buttons should generate the two "control arrows," moa, mob. Finally, the double button should generate esc.

This allows setting the rule by mouse movements, sometimes a convenience.

Selecting a rule does not enter it into the CAM/PC's lookup tables: this must be done by pressing the `insert` button; not doing so allows browsing through submenus without making a commitment. To a certain extent, it also allows the transportation of rules between environments. The CAM/PC screen will blink as the table is deposited, which may be taken as a confirmation of the operation.

It remains to initialize the automaton's bitplanes, done by pressing `Y`; there is no obvious acknowledgement, but close observation will reveal a time-change when the operation is complete, and until then no further operation will be accepted. At present the only initialization is to 50% random density, but this could change if there were future demand. It is also possible to continue using a previously existing set of files, or to employ a previously prepared set provided that the names F01.PAT ... F12.PAT were used.

Typing `t` will show a mean field curve for the density of live cells in the second generation as a function of that in the first. For most rules, it can be expected that the equilibrium density of the automaton will be the fixed point of this curve, but strong exceptions exist among totalistic and semitotalistic automata. The monomial probability gotten from `u` can be of assistance in choosing a rule whose curve has a desired form.

The use of `t` is usually reassuring; if too many trials have been made and the screen becomes cluttered, `z` will clear it. This is distinct from the action of `delete`, which clears the CAM/PC's screen by setting all bitplanes to zero.

At this point, the only thing to do is to type `S` to obtain an ongoing evolution, or `s` to single-step the development. `S` can be halted by pressing any key.

10.6 Variations

Inasmuch as the limiting cases in which a higher dimensional automaton approaches one of lower dimension through very short periodicities in one or more directions are important, provision has been made for a "cube" of few planes.

The options `h`, `i`, `j`, and `k` create lookup tables in the CAM/PC appropriate for a cyclic automaton of spatial period 1, 2, 3, or 4, respectively. These tables are distinct from each other and from the one generated by `insert`, but nevertheless displace it and persist until they themselves are displaced.

Since the run cycle of the full automaton involves shuffling disk files, in the exceptional case of CAMETD there are two ways to initiate an evolution; `S` and `s` refer to the full automaton as is the norm in CAMEX, but `G` and `g` refer to the four thin cubes which are entirely self contained within the CAM/PC.

The reflexes which one tends to establish while using CAMEX will cause much anguish and vexation if the two activators are confused (pressing `S` without thinking tends to become habitual). Also, the possibilities inherent in starting an evolution from the main menu after having left the submenu can be either a curse or a blessing, accordingly as the differences in their operation are understood.

Because subtleties of color are such an asset in visualizing the three dimensional automaton, especially given the absence of other clues, the keys `f9` and `f10`, which adjust color and allow CAM-plane viewing, have been brought over into the submenu.

An additional key, `f8`, has been endowed with similar properties relative to the planes stored in the diskfiles, so that they can ve viewed while the automaton is halted.

Of course, one single plane can always be emphasized, to the exclusion of the rest.

11 The Margolus neighborhood

One of the design objectives of the CAM hardware seems to have been to implement a particular structure amongst two dimensional automata known as the Margolus neighborhood. This neighborhood is of theoretical interest because it lends itself to the construction of reversible automata, and is often used for the simulation of lattice gasses.

11.1 The neighborhood

The Margolus neighborhood partitions both space and time into even and odd components. First, a mapping is defined for a 2×2 neighborhood into itself; in the CAM context, each cell has four states, producing 256 neighborhoods for each cell. The mapping is therefore defined by a 4×256 matrix.

Next, non overlapping neighborhoods are used to tile the plane. It is convenient to locate the lower left hand corners of the neighborhoods at lattice points both of whose coordinates are even, leaving the mutually odd coordinates to occupy neighborhood centers.

Since the neighborhoods do not overlap, there is no interaction between them; to create an interaction, time is also partitioned into even and odd moments. As time progresses, neighborhoods are alternately selected from even and odd coordinates, so that the neighbors which a given cell sees vary according the parity of the moment.

In principle, larger neighborhoods with longer timing cycles could be constructed; however such complexity lies beyond what CAM hardware permits. Furthermore, the hardware switching arrangements permit ignoring either the horizontal or the vertical parity, but this should probably be considered as a pathology which would not normally arise.

11.2 Rule of evolution

The fundamental construction is a mapping of a square, consisting of four cells, into itself; in each halfCAM the cells have four states, gotten by combining the two binary bitplanes. Each cell therefore requires a function of four four-state variables to describe its evolution. One way to represent the situation would be to use four 16×16 tables. The result is more manageable than using a single 4×256 matrix, but still requires the manipulation of quite a bit of data.

As with other automata, selections can be simplified by introducing totalistic rules or other specialized combinations. However, the Margolus neighborhood seems to have been designed with the specific intention of emulating a lattice gas, and moreover, reversibly so.

Therefore, when the CAM is operating in Margolus mode, the lookup table in the CAM as well as the circuitry required to implement it, has been arranged to favor this particular application. To the user, it appears that the same rule by which one single cell evolves from four neighbors has been applied differently according to the parity of the coordinates, albeit in a very symmetric way — by rotating the neighborhood.

Just for that reason it is recommended that the neighbors not be thought of as lying to the north, south, east, or west; rather they are situated clockwise, counterclockwise or diagonally opposite, all relative to the cell itself which occupies the corner required by parity.

If this were all there were to the matter, rules could be defined with great ease. Indeed, it is a good arrangement for creating

$(4, 1/2)$ Moore automata, giving each corner a rotated version of the same rule. The rule need not always be the same, of course.

However, as originally conceived, the CAM lookup table had twelve address lines, whose meanings could be varied somewhat by both software and hardware switches. Without worrying about variants, for von Neumann neighborhoods, five neighbors with four states (two bits each) in one half-CAM plus the center cells in the other half-CAMfills the quota. For Moore neighborhoods, nine cells in one plane together with three more center cells gives the pertinent dispensation of twelve bits.

Margolus neighborhoods require only eight bits to achieve four states in each of four cells, leaving four bits free. These free bits can be assigned to two parity bits (four parity combinations) and two phase bits, giving four distinct tables (phase bit combinations) each of which distinguishes the parity of the coordinates in the square sublattice. The tables can be completely separate, rotated versions of a single table, or a combination of the two.

But the additional addresses could also have been given over completely to the phase bits, leaving sixteen different tables to be applied uniformly to all the cells.

Finally, custom configurations can be brought in by invoking the user options and hardware jumpers.

11.3 Four tables for four parities

The table in Figure 7 shows how the bits in the Margolus neighborhood are used to index the lookup tables that will define the rule of evolution.

Following the general custom by which rule tables are defined, the states of the cells are regarded as digits to be arranged in some

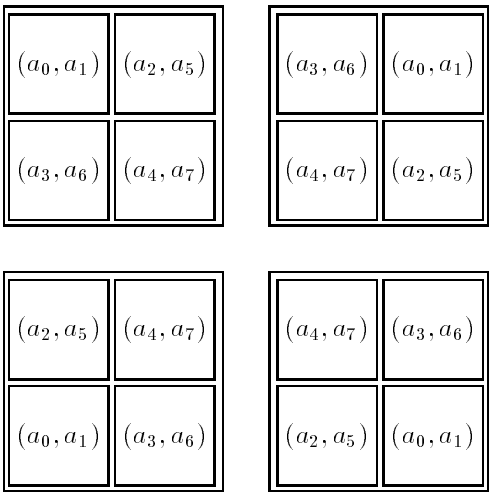

Figure 7: Margolus neighborhoods by parity

standard order; Figure 7 establishes that order for the Margolus neighborhood. The states have internal structure, so plane 0 is taken as providing the low order bit, plane 1 the high order bit for a cell of four states; this is acknowledged by filling the table with pairs.

Each different number corresponds to some distribution of the states around the neighborhood; to form the lookup table that is the number indexing the array in which the new value of the state for that particular neighborhood is recorded.

The "central" cell occupies a different corner of its neighborhood according to the parity of its coordinate, which is the reason that the table contains four versions of the arrangement of the neighborhood.

With a view toward constructing rules for reversible evolution, one of the restrictions could be that the new states be a permutation of the old ones, given the understanding

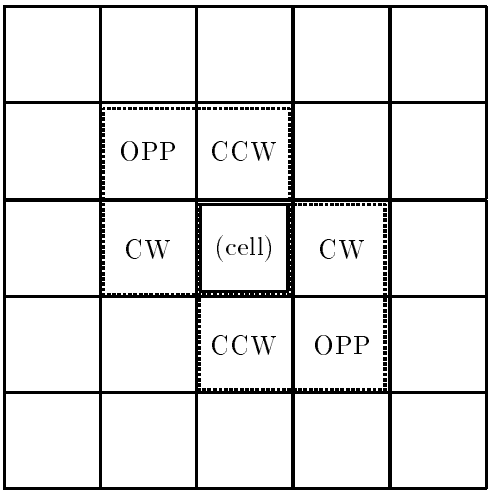

Figure 8: Parity can be reversed between generations

that permutations themselves are reversible; there are 120 permutations of four objects.

The layout of the screen panel is fairly typical, but so far has not been subject to the pressures and accommodations of prolonged usage.

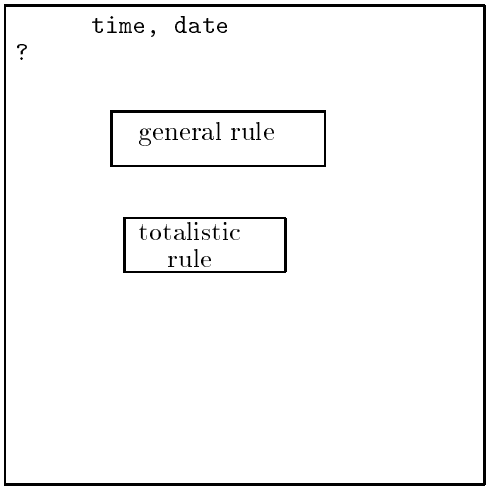

Figure 9: Margolus submenu

12 REC programming

First, REC stands for *regular expression compiler*, a rather fanciful term concocted one day when the program had to be given a name. Anyway the idea was to make programs follow the style of regular expressions, using elements of concatenation, selection of alternatives, and iteration. It was created for the purpose of completing an acceptance test for a PDP-8 (and consequently had to be very compact) on the basis of earlier work with "operator predicates" in LISP.

12.1 The language

The essence of REC is the use of four symbols of control (the two parentheses, colon, and semicolon) together with operators and predicates. Operators are just subroutines; predicates have a truth value in addition.

As usual, a pair of balanced parentheses is used for delimiting groups of symbols, but the group is also to be regarded as a predicate whose truth value depends on how its execution was terminated. Execution proceeds from left to right in the normal manner of reading English text, although the sequence can be interrupted by predicates, colons, and semicolons; all REC programs must be parenthesized.

A colon signifies repetition from the opening left parenthesis of any given level, being the embodiment of iteration in REC. On the other hand a semicolon implies proceeding at once to the closing right parenthesis, while assigning the value "true" to the process just completed. In point of fact one does not include the right parenthesis as part of the departure, but continues from the symbol just beyond it; meeting a right parenthesis as a normal part of a sequence also terminates the subexpression, but assigns it the value "false."

This leaves the role of predicates in a REC expression to be explained. Regular expressions themselves are indeterminate, since they describe the whole class of strings obtained through arbitrary choices of alternatives and iterates; a computer program requires a concrete choice at each juncture. Predicates provide the decision; when true the sequence of symbols continues without interruption, but falsity implies skipping to a new program segment.

The new segment is the one which immediately follows the nearest colon, semicolon or right parenthesis at the same parenthesis level. In the case of the parenthesis, the value "true" is assigned; this is the mechanism through the boolean complement of a truth value may be achieved. In other words, if `p` is a predicate, `(p)` is its logical negative. Likewise, when `p` and `q` are predicates, `(p;q;)` corresponds to `p` OR `q` and `(pq;)` to `p` AND `q`.

Arbitrary boolean combinations are possible; for example EXCLUSIVE OR corresponds to `(p(q);q;)`, but one must beware that `q` could get executed twice without being reproducible (for example, by reading the keyboard twice). Such dilemmas are not frequent, but they do occur and lead to schemes for preserving information, defining variables, and what not; none of these are dealt with at the level of skeletal REC programming.

12.2 Defining programs

Within the basic skeleton the programmer must provide for the individual operators and predicates, generally through a symbol table indexed by ASCII characters. REC is not required to work with single character symbols, but the complexity of parsing a program in the compiler is increased by a whole order of

magnitude if anything but the simplest variants on this restriction are allowed.

Over the years extensions to REC have occurred, and specialized applications have developed. The most important extension provides for creating new subroutines. They are designated by a composite symbol such as `@s`; "at sign" joins the list of reserved letters. Subroutines are defined by enclosing a list of definitions (followed by a main program) within braces:

```
{(...)  s (...  @s ...)};
```

in fact such a braced list may be included anywhere in a REC expression that operators, predicates, or subexpressions can occur. The scope of the definitions lies exclusively within the braces; the same symbols can be reused elsewhere or even recursively.

Certain additional features have proved convenient, but are common to most programming languages. One is to ignore space, tabs, and so on unless they are an actual part of quoted data; the other is to permit comments. In REC, comments are enclosed recursively within square brackets, which permits disabling debugging or trial segments without removing them completely.

It is hard to work with programs which are incapable of providing data to themselves; some form of quotation usually has to be provided. In some contexts, recognizing numbers as such and transforming them into a binary representation is essential; in others quoted character strings suffice.

A counter, written `!n!`, is a useful predicate, true `n` times and then false; `(!12!x:;)` would perform the operation `x` a dozen times.

The two most important levels of specialization that we have worked with include, 1^{st} introducing a pushdown list and, 2^{nd} introducing a workspace.

The whole mechanism of numerical processing goes well with the first; one predefines arithmetic operations, comparisons, input and output conversion, and so on, as appropriate operators and predicates. Fixed subroutines attend to such things as square roots or trigonometric functions, just as is done in other programming languages. When specific calculations are required, they are performed by writing programs.

Dealing with arbitrary arrays requires further machinery; consequently an arithmetic REC has a closer affinity to APL than to FORTRAN. Nevertheless fixed arrays are convenient, from which a matrix version of REC can be devised.

The second extension really produces an exotic structure, suitable for editors, compilers, and the like. However, both of these versions of REC are much more ambitious than what is included in CAMEX.

12.3 Operators and predicates

When it comes to choosing operators and predicates for CAMEX, a week's experience has hardly produced a definitive list; moreover there is a certain expectation that individual users of CAMEX will generate their own. This, of course, will lead to a proliferation of nonstandard variants, but so far that hasn't happened, even for the C programs.

Part of the need for something like REC comes from the proliferation of options in a program, and in finding some way to deal with them. Considering only LCAU's main screen, there is the full 64-neighborhood rule to be dealt with, then totalistic rules, product rules, and several kinds of reversible rules. Besides that, there is a collection of demonstration rules intended to show the user something of the variety which is possible, not to mention having selected them in advance for their attractiveness.

13 Edit a Moore plane

For many experiments with automata, it is sufficient to generate random bitplanes, or to fill the bitplanes with such simple designs as lines or squares. Chaotic evolution determines which automata are the most suited for this treatment, but the Zhabotinsky reactions, for example, evolve around a deterministic nucleus which occasionally needs to be explicitly inserted into the bitplanes.

Beyond that, there are automata such as *Life* and *WireWorld*, whose initial configurations need to be laid out very carefully and explicitly. These requirements are met, for (2,1) Moore automata, by the module CAMEML, composed of facilities for editing bitplanes, with emphasis on plane 0.

13.1 Outline

There are three aspects to editing the field of a cellular automaton. The most obvious consists of inserting, removing, and modifying the cells; actually the number of cells has to remain the same, so that these activities refer to live cells in the context of a binary automaton.

As to the second aspect, for a fixed rule, editing will most likely be used to obtain configurations following a desired course of evolution. This makes it convenient to be able to save and restore configurations, as well as to conduct trial evolutions, all in addition to setting up the configurations themselves.

Finally, although it might seem to be strictly a question of editing the rule table, one often wishes to create a rule causing a given course of evolution. This involves an interaction between observing trial evolution in the field of the automaton and the definition of the transitions in the table. A display of the full rule table is not necessarily required, but there must be sufficient access to be able to modify it.

13.2 Keyboard options

The complete alphabetical list of options needs then to be broken down into related groups for further explanation. In doing so, the console monitor should be distinguished from the CAM/PC monitor. The console will show an 11x11 patch taken from one of the bitplanes, accompanied to its right by a smaller 9x9 patch in which the second generation of an evolution can be shown. At the far right is a large panel which can show either help information or the full rule table, according to some of the options.

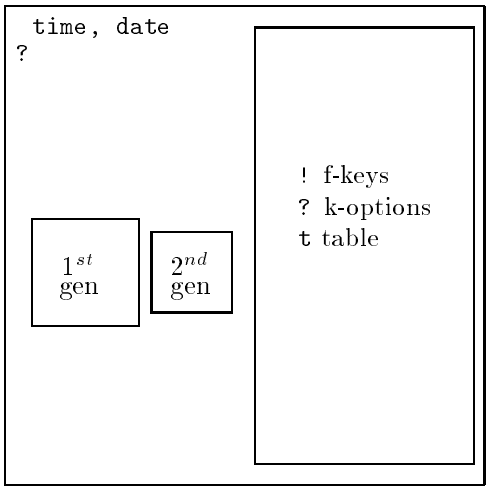

Figure 10: CAMEML console panels

If the recommended color combination is followed, and plane 0 is selected, the CAM/PC monitor will show plane zero in terms of white dots on a black background. Plane 1 can be used for a cyan echo, if the rule under investigation calls for one. Plane 3 carries a cursor, an 11x11 blue square with the same

dimension as left console panel, to which information can be passed back and forth.

Each panel carries a cursor, which can be moved by the mouse, keyboard arrows and symbols, or some of the function keys. The lack of a mouse will not result in the loss of any options; equivalent keys always exist.

13.2.1 Alphabetical listing

The full alphabetical list of the keyboard options, with a brief descriptor for each follows:

0,1,. - enter state for 1st generation
a,b - enter state for 2nd generation
& - AND the patch to selected plane
f - fetch demonstration
g - one cycle in console panel
G - back copy from 2^{nd} gen to 1^{st}
h - patch cycle (combine g, G)
H - repeating patch cycle
i - insert beneath cursor
I - insert whole patch
L - install Life w/echo
m - mark neighborhood
M - mark over whole patch
n - unmark neighborhood
o - OR the patch to plane epl
p - random plane
Q - square symmetry op
r - read patch
R - reset screen from disk
s - single step on CAM-A
S - execute on CAM-A
t - show rule table
u - fill panel patch with ones
v - mark neighborhood
w - put patch in CAM-A
x - uncommitted transitions
X - random table
y - random plane
Y - random plane 0 (50%)
z - clear panel patch
Z - clear rule table
? - operator help menu
! - function help menu

13.2.2 CAM/PC evolution

In keeping with the uniform practice for all CAMEX demonstrations, the letters `s` and `S` are reserved for cycles of the CAM/PC; by the same tradition, `s` advances just one single generation per keypress. In turn, `S` initiates free-running evolution which can be stopped by pressing *any* key.

However, care must be taken as to which rule governs the evolution. To permit browsing, and the transport of a rule from one demonstration to another, all of them require an explicit act — usually depressing INSERT — to install the CAM/PC's lookup table. Activating INSERT from within CAMEML (or generally by using INSERT in the main menu after having selected the random (2,1) Moore option) will install the rule currently resident in `mogrul` in plane 0, an echo for plane 1, leaving the possibility for a von Neumann shift rule to be installed in CAM-B.

From time to time specific automata are assigned to options (such as `L` for *Life*) which will install a particular rule. Beware setting the CAM in motion without having chosen the correct rule somewhere; no physical damage will result, but a carefully prepared pattern in the bitplanes may inadvertently be disrupted.

13.2.3 Console panel evolution

The consequences of evolution can be viewed in the console panel; the option `g` causes the second generation to appear in the small panel. It is small just because the console panel is not assumed to be cyclic, and so one cannot know the descendants of the edge cells. The table `mogrul` defines the evolution, but there are no further side effects such as occur in the CAM, there being no additional planes.

The contents of the small panel can be returned to the large panel by the option `G`

(which is *not* a repeating `g`), for yet another generation of evolution. The option `h` combines `g` and `G`, repeating indefinitely when `H` is used. The process makes the most sense when the pattern in the panel is isolated from its surroundings, or a still life is under investigation, or some similar circumstance.

13.2.4 Console panel symmetry

The console panel is square, consequently susceptible to the performance of symmetry operations. The ASCII arguments of the option `Q` result in the following operations; in all cases the symmetry axis passes through the central cell, which is likewise the center for rotations:

H - horizontal reflection

V - vertical reflection

D - diagonal reflection

A - antidiagonal reflection

L - rotate counterclockwise

R - rotate clockwise

I - inversion

N,S,E,W - cyclic directional shift

13.2.5 Data exchange

Information can be exchanged between the CAM area in the bitplane marked by the cursor and the console panel. Reading the bitplane is accomplished by `r`, whose counterpart is `w`. Using `w` replaces the entire patch, but when additional live cells are to be added without removing those already present, the option `o` (boolean OR) should be used. Boolean AND to the plane `&` is less useful, nor are there presently boolean operations running in the opposite direction.

Bitplanes can be built up by gradually introducing patterns into the console patch which are then transferred to a bitplane. The numerals `0` and `1` deposit their values at the cursor location, which then advances cyclically to the right; a dot has the same effect as a `1`. For purposes of generating a forced evolution, the keys `a` and `b` will insert zeroes and ones, respectively, in the second generation panel, at the location of a matching cursor which does not advance.

The console patch can be cleared by the option `z`, or set (to all 1's) by the option `u`. To clear all planes in the CAM, use ERASE.

Keyboard arrows, which should be assigned to the corresponding mouse movements, can be used to move any of the cursors; more details are presented in the section on mouse movements.

13.2.6 Rule table

Among the many motives for editing a rule table one finds the desire to enforce a desired course of evolution, transferring the problem to the domain of plane editing. This is the reason for two separate generation panels in CAMEML, together with several associated options.

Option `t` displays the rule table in the area of the help panel, together with a cursor reflecting the neighborhood surrounding the cursor in the first generation panel. Since we are editing a plane, not the table, this cursor cannot be moved on its own account; it always follows the neighborhoods. Nine cells imply 512 distinct neighborhoods, which are shown in two groups, according to the central cell; that implies two 16x16 tables, which are shown side by side.

Once the first and second generation patches have been edited for a desired effect, the option `i` will record the transition at the site of the cursor in `mogrul`. Similarly, `I` will record the transitions implied by the

full panel; the order in which it is scanned determines the resolution of conflicts should there be inconsistencies.

To facilitate the handling of possible conflicts, it is possible to mark certain transitions, so that they will not be redefined. This is the purpose of the table `mogaux`, whose existence is invisible to the user. Option `m` will mark a single neighborhood, `M` causes wholesale marking for the whole panel, and `n` can remove any single mark. All of them can be removed, to start over again by selecting `x`. (This use of `x` is consistent with line editing in LCAU, but not with its use to generate random rule tables, another important convention).

If a rule table becomes hopelessly befuddled, clearing it by `Z` gives a fresh start. The option `v` is similar to `m`, but marks the whole Golay class of the neighborhood (that is, the neighborhood and all its symmetry images).

Colors have been chosen for the tables, cursors, and markers with the intention that they should be pleasant and distinctive. However, their assignments can all be changed from the main menu; a combination of cursor options and function key `f7` suffices.

13.3 Mouse movements

There is great convenience in being able to manipulate a program by running a mouse back and forth along a table (or mouse pad), although the mouse can easily become lost in a screen full of icons. Moving cursors around, and tracing designs on the screen, are among the less controversial applications of a mouse.

13.3.1 CAMEX mouse

CAMEX uses what may be called an "external mouse," in contrast to an "internal mouse" whose control program would form an integral part of CAMEX. This means that the mouse program is a TSR (terminate and stay resident, in the language of operating systems), inserting its signals into the keyboard buffer just as though the user had typed them.

For compatibility with CAMEX, the following assignments should be made:

MOU - mouse up - UP ARROW

MOD - mouse down - DOWN ARROW

MOL - mouse left - LEFT ARROW

MOR - mouse right - RIGHT ARROW

MOA - left button - CTL LEFT ARROW

MOB - right button - CTL RIGHT ARROW

MOX - both mouse buttons - ESC

In recognition of the fact that many users of CAMEX may not have a mouse, some of the function keys have also been assigned to mouse movements. They have fixed responses, whereas the mouse itself has three different classes of movements.

13.3.2 With or without a mouse

Initially, the mouse guides the cursor in the console panel, but MOX will replace the cursor by the illuminated patch on the CAM screen. Initially it moves by its own width, but a second use of MOX reduces the displacement to a single pixel. The cycle is of length three, returning to the console cursor.

Likewise the response of the buttons varies with the class. They toggle the pixel in respectively the first generation panel or the second when in console phase; in CAM phase, they read or write the cursor patch, respectively.

A handful of positioning options are related to the mouse movements, but only available as keyboard options for the console

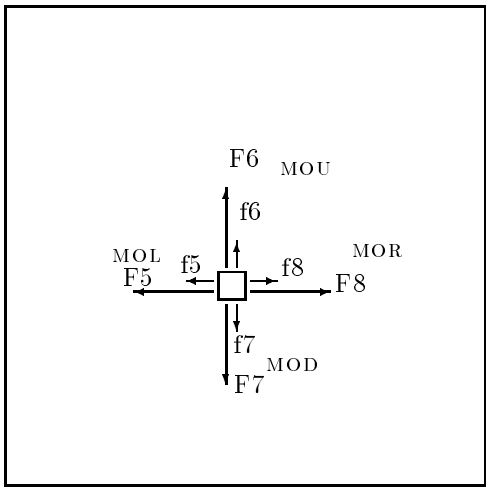

Figure 11: Movements of the illuminated patch on the CAM screen. They can be generated by mouse or by the keyboard.

panel. They position the cursor at the margins, but they are not much used.

13.3.3 The mouse menu

The following list, which is rather redundant, is also illustrated in Figure 11.

MOD - downward

patch - cursors south
long - full patch south
short - patch 1 pixel south

MOU - upward

patch - cursors north
long - full patch north
short - patch 1 pixel north

MOR, SPACE - forward

patch - cursors east
long - full patch east
short - patch 1 pixel east

MOL, BACKSPACE - backward

patch - cursors west
long - full patch west
short - patch 1 pixel west

MOA, CTL LEFT ARROW -

patch - toggle 1^{st} gen cursor cell
long - read patch from CAM screen
short - read patch from CAM screen

MOB, CTL RIGHT ARROW -

patch - toggle 2^{nd} gen cursor cell
long - read patch from CAM screen
short - read patch from CAM screen

MOX, ESCAPE - advance mode phase
< - cursor to left margin
> - cursor to right margin
^ - cursor to top margin
_ - cursor to bottom margin
* - cursor to center of field

13.4 Function keys

The function keys are used to execute a REC program, move the cursor patch, or manipulate the colors of the planes. Some of the items repeat information given in Figure 11.

13.4.1 Full list

In its entirety, the list is:

f1 = load bitplane(s)
f3 = edit rec program
f4 = execute rec program
f5 = patch west
f6 = patch north
f7 = patch south
f8 = patch east
F5 = patch long west

F6 = patch long north

F7 = patch long south

F8 = patch long east

f9 - show next plane

F9 - primary plane colors

CTL f9 - show plane 0 only

ALT f9 - show previous plane

INS - center cursor

DEL - clear all planes

f10 - xor patch

F10 - erase patch

13.4.2 Load bitplanes

Random bitplanes of various densities and designs suffice for experiments with the majority of automata; there are a few, such as *Life* and *WireWorld*, which often require elaborately prepared bitplanes for their original configurations. One way to obtain them is to include generators for specific patterns in the submenu for that automaton, or to creat the pattern by using the editor.

Once created, it is preferable to save the pattern, rather than having to generate it anew in a future session; conversely, it is desirable to load bitplanes which have been prepared previously, whatever their source.

The option `f1` scans the current directory for any files with the extension PAT, listing them on a newly cleared screen. Since the symbol Y is used elsewhere in CAMEXto create bitplanes, it can be used here with a hexadecimal argument to load the file marked by the cursor into that combination of bitplanes.

Where there is no uncertainty, INSERT will load plane 0. To load multiple planes, usually from separate files, they should have names which embed the argument for `y`. Such a convention seems better than creating a plethora of extensions.

13.4.3 REC program

The easiest way to move information back and forth from the CAM/PC to the disk units is to write a small REC program, given that a full path name can be inserted as a string constant via REC, but is less convenient to do otherwise. Therefore the main menu's keys, `f3` for editing a rule, and `f4` for executing it, are included in the CAMEMT menu.

This is the procedure to be followed to save a plane or to load it from another directory; the option `f1` simplifies loading from the current directory.

13.4.4 Patch movement

Given that the arrows are attached to a mouse in a three phase cycle, some function keys have been assigned the task of moving the patch around the bitplane independently of the cycle. Not only is their assignment unambiguous, it is always available, even when the mouse is not. The key sequence is somewhat mnemonic, westmoves getting the leftmost key and so on.

Some keys toggle the patch (`f10`), or initialize it to the center of the screen; otherwise it retains its position from one usage to another, even when CAMEML is abandoned and reentered. It is a general characteristic of CAMEX menus that the position of their cursors is retained through an entire session.

13.4.5 Color, plane

The keys `f9` and `f10` have been reserved throughout most menus to permit visualizing the planes one by one, choosing a standard color assignment, or stepping through a sequence of color assignments. Likewise, ERASE almost always erases the full complement of CAM bitplanes.

14 Conway's *Life*

One of the most famous of all two dimensional automata is the game of *Life*, invented in the late sixties by John H. Conway, and publicized in Martin Gardner's monthly "Mathematical Recreations" column in *Scientific American*. Several of the demonstrations in CAMEX are based on *Life*; along with the program some data files are provided to load the bitplanes with interesting initial configurations.

14.1 Description of *Life*

Life requires a two-dimensional binary Moore neighborhood; therefore neighborhoods containing a central cell with eight additional neighbors. The rule is semitotalistic, meaning that the state of the central cell together with the number of remaining live cells determines the evolution. In these terms a live cell (binary state 1) lives in the next generation whenever it has two or three live neighbors. The quiescent state (0, also called dead) becomes live (a new cell is born) whenever it has exactly three live neighbors. All other cells die (or remain quiescent).

This automaton is remarkable for the fact that an initial population of randomly chosen live cells eventually settles down into a collection of visibly separated objects which run through short cycles of evolution. The most common cycle, of period 1, is called a "still life" but there are "oscillators" and "alternators," mostly of period 2. Period 3 objects are quite rare; occasionally others are found with longer periods.

One especially striking five-cell object, called a glider, translates itself diagonally by a single cell every four generations. Its phases comprise two pairs of mirror symmetric figures; the name is therefore a pun on the crystallographic concept of a glide plane.

After a long evolution, small residual objects are the most common; a graph of frequency versus size is instructive. Automata other than *Life* tend to evolve into uniform chaotic fields of a fixed density, or to dwindle away altogether. *Life*-like rules are merely uncommon, not unknown; hard it is, however, to find another rule exhibiting the organized behavior discovered within *Life* — certainly none governed by a rule of equal simplicity.

14.2 *Life* artifacts

The earliest analyses of *Life* involved following out the evolution of fairly simple clusters of live cells manually, but computer simulations of random fields as well as deliberately chosen patterns also revealed many characteristics of *Life*.

14.2.1 Small objects

It was quickly observed that some configurations are quite stable, others highly volatile, and naturally a considerable range in between. No persistent objects formed from one or two cells exist. Three cells either horizontally or vertically aligned, called *blinkers*, alternate from one form to the other; they form the class of permanent object with the tiniest member; nevertheless they are among the most numerous.

Among four-cell objects, small rings and especially squares are both numerous and stable. Squares are often overcome by expanding clouds of cells from a nearby region and yet survive when the activity has subsided. As a result they are often deliberately placed within a construction to establish boundaries or reference points. Another seven-cell object, called an "eater," has been

found useful for similar purposes.

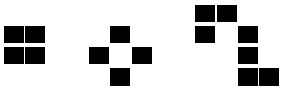

Figure 12: Some small stable *Life* objects

Besides static objects, or *still lifes*, there are some small oscillators which are frequently observed after a random initial field finally settles down. Blinkers are ubiquitous, but from time to time others may be observed.

Figure 13: Commonly seen period 2 objects.

14.2.2 Gliders

Their small size makes gliders a frequent intermediary in evolving configurations; it is also understandable that glider collisions were among the first *Life* processes to be studied. It was found that glider collisions of varying complexity could produce all kinds of stable objects as an end product.

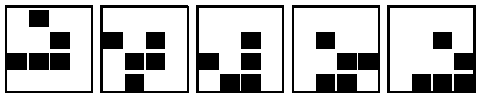

Figure 14: A glider advances one square diagonally in four generations.

14.2.3 Volatile objects

In the realm of volatile objects, several small configurations were found which were capable of expanding hundreds or even thousands of generations before settling down into a stable ending. One of these, of seven cells, resembling the greek letter Π, occurs frequently in general evolutions; its second generation, sometimes called a "thunderbird," is also common. Not only are they of frequent occurrence; their expansion is quite aggressive, although the region of evolution usually collides with other regions in the field and soon loses its identity.

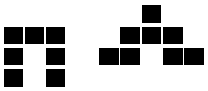

Figure 15: First and second generations of a very volatile object.

14.2.4 Oscillators

By mixing stable and volatile artifacts, many investigators produced a huge variety of long period oscillators; some of them were capable of creating gliders when placed in close proximity so that non-essential debris in their cycles could interact. The culmination of this effort was a remarkable series of constructions and constructors.

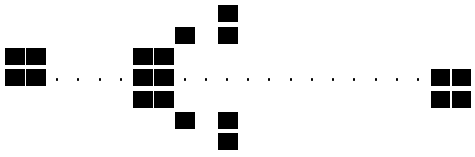

Figure 16: An oscillator of period 30.

14.2.5 Glider guns

Glider guns were the most straightforward; they were oscillating configurations which continually generated gliders, solving one of Conway's questions: Is there a *Life* configuration for which the number of live cells can be demonstrated to increase without limit? They also make good *Life* demonstrations. A glider gun is a little larger than is convenient to illustrate here, but the principle behind it is not too hard to describe.

In addition to the period 30 oscillator shown in Figure 16, there is another of period 46 which has been discovered, both of which involve a central seed which expands into a cloud which just happens to leave behind a copy of the original seed facing in the opposite direction. Naturally this tends to repeat the cycle except for the interference caused by the remaining products of the evolution.

It is at this point that placing stable elements nearby helps to create an oscillator; but if two oscillators are placed side to side, for the period 30 oscillator, or at right angles, for the period 46 oscillator, the debris may interact to produce a glider. For all this to work it is evidently important that the glider be small, and that small objects be of frequent occurrence. Even so, careful adjustment is required, and at that only one or two relative positions results in gliders which can break free.

14.2.6 Puffer trains

There are two classes of artifacts which grow without limit, in the sense that it can be demonstrated that the number of live cells in the field will always continue to increase. The first to be discovered consisted of the glider guns, for which a continuous stream of gliders emerged from a stationary region; the readily apparent periodicity of the process constituted the element of proof required.

An alternative would be for a moving source to leave behind a trail of stationary objects. Gliders are too small and fragile to serve the purpose, but other artifacts are suitable. *Space ships*, as they are called, are seen, albeit infrequently, in the residue evolving from random initial configurations; they differ from gliders by moving along the coordinate axes instead of diagonally. Nevertheless, they involve a reflective stage, managing to advance by two cells every four generations.

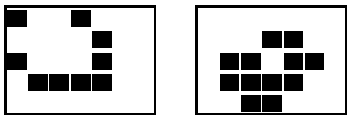

Figure 17: A space ship that moves two cells to the right in four generations, shown in two of its four phases.

In common with complex oscillators, there are some sparks associated with the movement of a space ship which are capable of interacting with other objects without impeding the motion of the space ship. Quite a large variety of followers can be constructed, many of which leave an orderly assemblage of residues in their wake; this is the combination which is called a *puffer train*.

There are even assemblages which leave no residue — smokeless puffer trains — which are really space ships themselves; the majority leave an untidy mess which may even take a life of its own, advance rapidly, and destroy the engine.

14.2.7 Other constructions

The description of all this goes considerably beyond the present discussion; however by using the bitplane editor and an appropriate reference (or even ones own imagination) it is possible to make some very interesting discoveries about *Life*; it is still not beyond the realm of possibility for them to be new and original.

Still, some of the really advanced concepts, such as discrete logic elements, a universal computer, or a universal constructor, require far more space and time than available in a simulator such as the CAM/PC(or any other system of modest size); even so many of their essential constituents can be tried out.

14.3 *Life* experiments

The main menu, displayed by the `f1` button, begins with four *Life* demonstrations. The first is a regular *Life*, executed in plane 0 with a glider trace in plane 1; this is possible because the five-cell gliders all fit within a single Moore neighborhood, to be detectable by a suitable rule for plane 1. As the alternate rule in plane 1, all the cells that were ever alive can be marked, visible as a halo around the evolving field.

Two demonstrations are not really *Life* demonstrations at all, but a pair of reversible rules derived from *Life*.

The fourth demonstration employs plane 1 to inhibit evolution in plane 0; among other things, this allows life to evolve within fixed boundaries. Choosing a checkerboard for the other plane gives nice results.

The REC menu offers much the same selection, with one addition - two-color *Life*. In this variant, live cells are colored, red and blue, say. The usual rule of evolution applies, without regard to color, but newly born cells take on the minority color of their parents whenever there is a choice.

14.4 Loading a *Life* plane

File management operators have been added to REC, through which the CAM/PC's lookup tables and/or bitplanes can be loaded. It is first necessary to define the file name, which should be stated as a string constant: for example, `"LIFE.PLA "`. The final space, terminating the string, is essential; any directory path acceptable to MS/DOS is also acceptable.

Once a file has been specified, the definition remains valid until another string constant appears in the program; this feature should be used cautiously. One of the following four operators may then be used:

x - load the lookup table from the disk
X - save the lookup table on the disk
y - `yp` loads hexadecimal bitplane combination `p` from the disk
Y - `Yp` saves bitplane `p` on the disk

These mnemonics have been chosen for compatibility with the menus, wherein `y`'s refer to bitplanes and `x`'s to rules. The same conventions regarding `p` also hold: for `y`, `p` can specify multiple planes (or none) by running through the range `0, ..., f`. Only single planes can be stored, so the range is `0, 1, 2, 3` in `Yp`.

With respect to REC, no file naming convention exists. However, some menus contain the function keys `F1` and `F2` which employ the CAM/PC conventions of `TAB` amd `PAT` as extensions for tables and planes. The files are interchangeable, with the exception that CAMEX has files of one single size (`4K` for `TAB` and `8K` for `PAT`) whereas CAM/PC has a greater variability. To load four distinct bitplanes in CAMEX, four files are necessary.

15 Zhabotinsky reactions

One of the most successful applications of two-dimensional cellular automata has been to the so-called Zhabotinsky-type reactions; their mathematical model has been extensively studied by J. M. Greenberg, S. P. Hastings, and others.

15.1 Theory

It is generally fruitless to model a differential equation by a cellular automaton with few states, but nonlinear differential equations exist which exhibit well-defined states with definite rules of transition between them.

15.2 General framework

Typically, a system will have three-state cells — resting, active, and exhausted. A certain number of active cells can activate a resting cell, which thereupon enters into the exhausted state for several generations. Once the recovery time has elapsed, a cell rests until activated anew.

Activity has many interpretations: initiation of a chemical reaction, infection with a disease, enthusiasm over hearing a rumor. Exhaustion runs a similar gamut: overheating or depletion of the reactants, incubation of the disease, boredom.

15.2.1 Spread of epidemics

The terminology of epidemics - infection, incubation, recovery - is convenient to describe the rules of this class. The models all follow the same general outline; there is always a healthy population which is nevertheless subject to infection at any time.

Infection through a single contact is the most plausible model, but one way to account for a disease which is hard to transmit would be to require a higher intensity of infection, exemplified perhaps by requiring multiple simultaneous contacts before infection occurs.

Once infection has occurred, nature must take its course. The model followed in this submenu supposes that a cell, once infected, itself becomes infectious through the duration of its recovery. Eventually recovery occurs, this model supposes no immunity, leaving the cycle free to repeat at any moment.

15.2.2 Neural conduction

Neurons, although one dimensional, are observed to follow the same three stages of activity. Basically, the neuron is inactive, but a stimulus can trigger a depolarization wave along the long tubular cell membrane. Slowly acting chemical processes restore the electrical potential across the cell wall, ready for another discharge.

Contact between neurons occurs at the far ends of the tubes - the synapses, various conditions governing whether a polarization wave crosses the bridge. The structure is neither as regular nor as local as a cellular automaton.

15.2.3 Chemical reactions

The Belousov-Zhabotinsky reaction takes place in a fairly exotic environment, since certain chemicals have to be chosen; one of them is an indicator, whose color changes permit the reaction to be visualized. The ingredients are the same as for the other examples; a system occupies a fairly precarious equilibrium in which an exothermic reaction can be triggered.

Once underway, the production of heat and exhaustion of the reactants eventually stops the reaction, which has to wait for a new supply of reagent to diffuse into the region and for the mixture to cool before starting

up again. The reaction occasionally becomes self-sustaining, with spiral waves radiating out from permanent foci.

15.2.4 Computer simulation

Several parameters are visible in the formulation of the automaton — the kind of neighborhood, the conditions for activation, the length of enforced inactivity. Neighborhoods for the CAM/PC are pretty well fixed; the Moore neighborhood in a single plane is convenient. The remaining planes are available as counters, giving up to eight generations of suspense; the rule for starting the count determines the activation criterion.

15.3 Zhabotinsky submenu

The menu for Zhabotinsky-like automata consists of two tables, differentiating between the treatment of active and inactive cells. The first defines the recovery of the cell, the second its activation. Each table is a matrix; rows are indexed by age or generation, (running from 0 to 7) while the columns are indexed by the number of active neighbors (running from 0 to 8). The possible rules are therefore semitotalistic, further modified by the counter.

Because there is more interest than the ordinary in varying the parameters in a Zhabotinsky rule, a large part of the main menu has been repeated within this submenu, together with help panels summarizing both the keyboard options and and function key options. The relevant keyboard options are those loading the bitplanes with designs or random numbers; the pertinent function keys are those governing the selection of colors and the examination of individual planes. Of course, evolution can be started and stopped in the usual way.

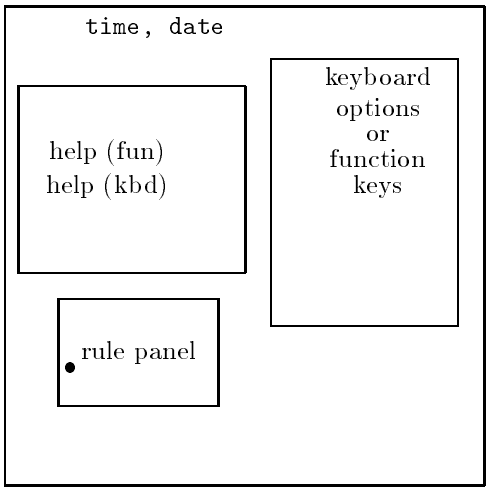

Figure 18: Zhabotinsky console panels

15.3.1 Keyboard options

In detail, the keyboard options are:

0,1 - define state
MOA - toggle state
SPACE, MOR - cursor right
BACK, MOL - cursor left
MOU - cursor up
MOD - cursor down
TAB - change center cell
INS - activate rule
DEL - clear all planes
c# - dense random plane
h# - 8x8 checkerboard
H# - 2x2 checkerboard
i - initial table
k# - checkerboard cell, age config
K# - checkerboard cell, age config
m - mark rule
M - unmark rule
p# - sparsely filled plane
R# - random plane
s - single step of evolution
S - generations of evolution
u# - fill plane with ones
w# - a few random points

x - random protected selection
X - random free selection
y - random plane
Y# - random plane 0
z# - fill plane with zeroes
! - help panel - function keys
? - help panel - keyboard

For keys requiring a numerical argument, this fact is shown by the notation #; the CAM plane identification scheme applies to the parameter.

The `k` options generate an age structure taken from their argument, placed in a checkerboard of the indicated size, the complement occupying the complementary squares.

15.3.2 Function keys

As with the keyboard options, the assignment of the function keys is summarized here for convenience.

`f7` - define parameters
`f9` - show next plane
`F9` - default plane colors
`ctl f9` - show plane 0
`alt f9` - show previous plane
`f10` - next color scheme
`F10` = natural color seq
`ctl f10` - first color scheme
`alt f10` - previous colors

Often a change of colors will enhance the appearance of the reaction, while the examination of individual planes can give some idea of the age distribution.

15.4 Suggestions

The parameters is a Zhabotinsky type reaction include the number of neighbors required to excite a cell, and the length of time it remains excited. The following suggestions help choose the parameters, thereby fixing the rule.

15.4.1 State 0

State 0 is normally quiescent, so that the entry for the zero state with eight zero neighbors should be zero. Zero states do not age, but there is a period of transition in which a cell that has just recovered still has the apparent age at which the recovery took place. That is because the counter in planes 1, 2, and 3 is reset by a zero in plane 0, or incremented by a 1. But the reset cannot take place until the cell has actually become zero, which is one generation later.

Otherwise the lines describing aged zero cells would be meaningless; to adhere strictly to the Zhabotinsky model, all such lines should be filled with zeroes. But they can also be used for other purposes if one wishes to think that a cell just recovered may relapse, starting to age anew, according to the condition of its neighbors.

15.4.2 Degree of contagion

The line for state 0, age 0, contemplates up to eight neighbors, various combinations of which can cause activation or infection. Having a single neighbor initiate the reaction may be more realistic epidemiologically, but it also leads to a high level of overall excitation.

Requiring a pair of neighbors to initiate an infection, it is harder generate activity with low densities of concentration, since the occurrence of pairs is much rarer than the occurrence of singlets. Once activation occurs, enough neighbors are usually excited all at once to become self sustaining.

Note that there is a difference between having an exact number of neighbors which will initiate an infection, and a threshold by which any larger number will also suffice. Epidemiologically the latter would probably be more realistic; the rule may be chosen to suit either alternative.

15.4.3 Initial density

When the reaction depends on the presence of a very small number of activated neighbors, the density of the initial configuration is fairly critical in determining the subsequent course of evolution.

This is the reason that the density parameter is made relatively accessible to the submenu, because its variation will produce the most instructive results. Therefore it is not nice to have to jump back and forth to the main menu all the time to adjust this parameter.

Either option `w` (few points) or option `p` (low density) can be adjusted to the threshold, in either case it ought to be rather low.

15.4.4 Length of dormancy

The right hand side of the rule table, the one that corresponds to state 1 in plane 0, can be filled out in many ways; all the entries determine whether a cell will continue to age, retaining its infectious property, or recover. As with the left hand side, the recovery can be made to depend upon the number of activated neighbors. Note that this particular programming of the CAM does not allow cells to be non-infectious and simultaneously non-infecting; a zero cell always resets the counter, leaving no way to determine its age.

In the classical rule there are three states: quiescent, infectious, passive, whose sequence is followed cyclically. The process is initiated once an infectious cell stands next to a quiescent neighbor.

Another interesting variant is to fill the tables with 1's; then place a zero in the 0 live-neighbor, 0 age position of the activation table, and a diagonal of zeroes in the recovery table. Option `i` performs the first part.

15.4.5 Speed of evolution

The speed of the CAM/PC is great enough that the evolution often appears as a blur; often single stepping is more informative. Also, the choice of coloring for the planes can vary one's perception of the evolution quite considerably.

In the worst case, plane 0 can be displayed all by itself, without any of the aging information. There is opportunity to do some further work with CAM's color map.

15.4.6 REC demonstrations

Several demonstrations, as well as REC programs, involve Zhabotinsky-like automata. The classical Zhabotinsky reaction goes through a three step cycle, usually producing concentric rings of epidemiologically excitation; two different frequencies arise under slightly different circumstances.

The key to a continuing excitation, manifested by spiral waves, and discovered by Greenberg and Hastings, is a central nucleus consisting of a closed ring of successive states. That allows a neverending sequence of activation and recovery, which propagates outward from the nucleus.

Several nuclei will produce interacting spirals. Very short loops, often just a collection of three cells, form the most common nuclei.

The REC demonstration must be interrupted twice, by a keypress, in each case preferably after the waves have stabilized; it is called "an interesting eater cycle" in the menu. Sometimes a "clean" demonstration results, in a way resembling the peeling of an onion. On other occasions some wild evolution sets in, quickly overpowering the orderly regime that was intended for the demonstration.

But that, in itself, could be considered as a demonstration.

16 "Eater" rules

There is a close affinity between Zhabotinsky type rules and the class of "eaters," defined as those for which the central cell takes on the identity of one of its neighbors. The usual arrangement is to rank the states in some order, then close the list to form a cycle. In a traditional form, "scissors cut paper, paper covers stone, stone breaks scissors." An automaton results when each cell suffers the ravages of any neighbor in a position to affect it. A stone cell becomes paper if there is a paper cell roundabout; meanwhile the paper cell suffers a fate dependent upon its own neighbors.

In the CAM/PC, the greatest variety comes from the von Neumann neighborhood, whose cells can have as many as four states; however this limits the number of neighbors to four. A much greater number of states produces a more interesting automaton, but four is still sufficient for many interesting studies.

CAMEX still does not have a submenu entitled "eaters," but the rule can be incorporated into just about any type of automaton which has a sufficient number of states; generally, the more the better. Consequently several of the submenus, for which this automaton makes sense, include options to generate the eater rule without having to edit the rule table.

The evolution of an eater rule progresses through stages.

Given a large number of states, there is a small probability that a cell will be affected by one of its neighbors. Initially the evolution consists of the formation of small droplets; whenever a cell actually changes its state, there is a chance that it can feed on neighbors which were previously unattractive. Similarly, the growth of any region consisting of a single state increases the number of additional cells which it might devour.

Once the droplets have begun to coalesce, two new things begin to happen; uniform regions start to collide, chains of cells close into cycles. The chain may even have been present initially; if the number of states is small, there may be nuclei where the states are clumped together.

Whenever chains become established, they generate spirals which sweep out larger and larger areas, because cells in the spiral having the appropriate phase will eventually pass by any cell sitting on the edge of the spiral and capture it.

In the end, regions which do not interact further will establish themselves, unless there are spirals present. A single spiral will go on spinning; with two or more, they can steal cells from each other, provoking a duel which will only end with the eventual dominance of one or the other.

On the other hand, they may just continue to coexist, circling around each other forever.

Introducing eater rules into cellular automata is more successful than trying to do such things as discretize Laplace's equation. In the latter case, the value (that is, state) of a cell should be the average of its neighbors.

Programming such a rule leads to an evolution which quickly settles down to an average value; if the number of states is small, it is impossible to get interesting evolutions in which boundary conditions are maintained. The reason is simply that few states cannot maintain a gradual variation between appreciably differing values on the boundary.

Applications to nonlinear differential equations, such as those that describe Zhabotinsky reactions, can be more successful whenever the solutions cluster around well defined values with definite criteria for the solution to jump between them.

17 *WireWorld*

WireWorld is an automaton which can be used for modelling digital logic in a plane. Three bitplanes are required, one of which bears a circuit diagram (the wires) and never changes. The other two are used to describe "electrons:" *two* in order to polarize the electron, forcing it to move along the wire.

17.1 Definition

WireWorld is equivalent to a four-state automaton with Moore neighborhoods, wherein the four states have the following significance:

0 - background cell
1 - wire cell
2 - electron head
3 - electron tail

The rules of evolution are:

- a wire cell becomes a head whenever one or two Moore neighbors (besides itself) are heads; otherwise it remains a wire cell.
- a head always becomes a tail
- a tail always reverts to a wire cell
- the background never changes

In a way, the rules resemble a Zhabotinsky reaction; an electron head is the activated state, the tail the refractive state, the wire itself constituting the inactive medium. The background confines the activity to the wires. Of course there is no requirement that the wires must be lines; something other than digital logic would result though, if they were not.

Constituting the electron as a pair, head and tail, polarizes the combination, forcing it to move in the direction of the head. Other mixtures would be possible, but are generally avoided as irrelevant to the task of simulating digital logic; likewise two electrons are not supposed to travel in opposite directions on the same wire. Nevertheless, arranging junctions where electrons can meet; that is, forming logic elements, is one of the design problems to be solved in *WireWorld*.

Normally the CAM/PC does not allow a four state Moore neighborhood, but the affinity between *WireWorld* and the Zhabotinsky reaction and its implied counter lets the rule be implemented. Plane 1 holds the circuit diagram, plane 0 the electron's head, plane 3 the tail. To enter the head state is the only decision which requires any neighborhood beyond the cell itself; assigning it to a Moore-neighborhood plane 0 fits just nicely into the CAM/PC's capabilities.

17.2 Digital logic

It is evident that the simulation of digital logic by a cellular automaton cannot proceed on the basis of voltage levels, because changes could only propagate from one place to another at a finite velocity. Something akin to pulses, which *can* propagate, is required; this supposes, in its own turn, the ability to distinguish between a pulse and its absence.

A traditional solution to the problem has been to maintain two lines, one for the signal and one for its complement. Additional states on a single line can accomplish much the same purpose.

Yet another solution has been to clock the signals; the complement of a signal is simply one which does not appear at the place and time that the signal itself should have done so.

In turn this can create the problem, either of synchronizing several local clocks initially, or distributing clock pulses emanating from a central source throughout the network. The

central clock would still have to be started, and allowed to run long enough to reach each region before the circuit as a whole would have to start operation.

Practically speaking, it is better to have local clocks, and to prepare the initial *WireWorld* configuration meticulously, with attention to the phase and period of each clock.

Another detail which must be watched, is to be sure that any two successive pulses are sufficiently well separated that they do not interfere with one another — especially when entering or leaving a logic element.

17.2.1 Barriers

The most fundamental object in *WireWorld* is probably a barrier — a circuit element which blocks pulses arriving from either direction. In and of itself, it does not amount to much; however it occurs frequently in the following constructions, sometimes participating in schemes circumventing it, sometimes for its ability to block interference.

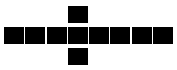

Figure 19: *WireWorld* barrier

17.2.2 Diodes

Another fundamental object in *WireWorld* the diode — a circuit element which will enforce the correct direction of travel on pulses — which often prevents spurious signals from leaking into a region. The diode shown in the illustration will pass signals from left to right, but not in the opposite direction. Its relation to a barrier is evident upon inspection.

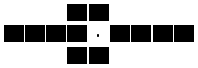

Figure 20: *WireWorld* diode

17.2.3 OR gate

A "t" can be used as an OR gate for a pair of signals; the main concern is that one signal should not encroach upon the territory of the other when the second is absent. The signals arrive along the arms of the "t," the composite departs along the stem. It is essential that the stem rise one cell above the crossbar, thereby isolating the incoming signals; a barrier lurks beneath it all.

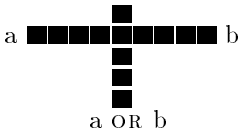

Figure 21: OR gate

17.2.4 EXCLUSIVE OR gate

A structure somewhat more complicated than an OR gate will produce an EXCLUSIVE OR. As a circuit element it is not much used in design, but it has a theoretical importance with respect to minimal collections of gates from which all others can be derived. For example, NANDs by themselves are sufficient for the purpose.

17.2.5 ONE-AND-NOT-THE-OTHER gate

Half of an EXCLUSIVE OR would be a gate of the form $a\bar{b}$. Theoretically it is useful because of the identity $ab = a\bar{b} \oplus a$; fortunately the

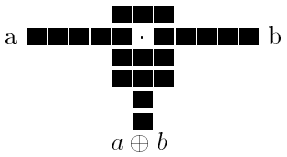

Figure 22: EXCLUSIVE OR gate

latter composite has an equivalent realization that is much less complex than following the formula literally.

Signals enter from the left and the right, the result appears at the bottom.

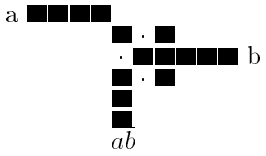

Figure 23: ONE-AND-NOT-THE-OTHER gate

17.2.6 AND gate

A little ingenuity produces the following AND gate; the signals enter at the top corners, their AND exits at the bottom right hand corner:

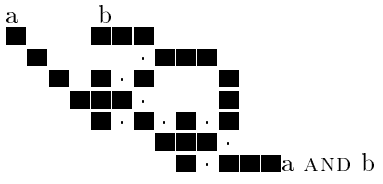

Figure 24: AND gate

This literal construction shows too many cells; the circuit can be rearranged into other forms.

17.2.7 Clocks

To establish a clock it is only necessary to set up a ring whose circumference has the desired period, draw the signal off at some point, and insert an electron. Diodes can be used to protect the clock, and to ensure a particular direction of circulation. More elaborate clocks can be constructed by spacing out several electrons within the same circumference.

WireWorld inherits the basic three-cell loop from the Zhabotinsky reactions, yielding all kinds of period-3 clocks. That is also the closest spacing that two electrons can have, but it is entirely too fast and dense for most applications.

A ring of four cells gives one of the smallest manageable clocks:

Figure 25: *WireWorld* clocks

However, loops yielding period six or even longer lead to more conservative constructions. Period six allows the inclusion of ultrafast period three subassemblies.

A clock seems to be the only way to create the boolean constant TRUE; once again it is worth emphasizing that the constant only appears at intervals and propagates with a finite velocity.

17.2.8 An inverter

The $a\bar{b}$ gate becomes an inverter when the a signal is always TRUE; this has to be arranged with a clock, simultaneously creating a timing which has to be respected. In particular, the period of the clock establishes the minimum interval between pulses.

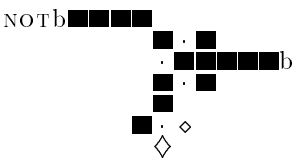

Figure 26: COMPLEMENT

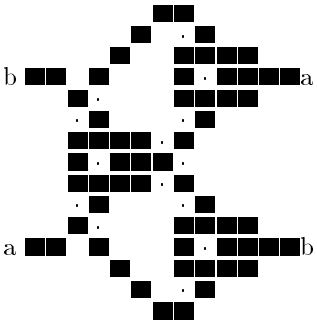

Figure 27: CROSSOVER

17.2.9 A crossover

Having exhibited the boolean operations, all the necessary material is present to construct a combinatorial circuit; by adding a unit delay, sequential circuits are also possible. Leading the signal along a zigzag is all that would be required, so there is no difficulty. To realize circuitry in a plane, however, crossovers are required.

One technique would be to multiplex a signal along a single wire, then demultiplex it with the help of a clock. Nevertheless clockless alternatives are available; one extravagant procedure is to use the identity $b = a \oplus (a \oplus b)$ to release b from an EXCLUSIVE OR in which a was the left member, and its counterpart to release a on the right.

17.3 Digital circuits

Once it is clear that all the components for digital circuitry are present in *WireWorld*, the only thing lacking to produce a design is to assemble them. Presumably the same course that would be followed with real components should be recommended. In other words, the next step is to create the traditional small components, such as the 7400 series of TTL integrated circuits. Not necessarily starting with the historical package of four NAND gates; but rather with such items as flip flops, shift registers, adders, memories.

At the same time, one should seek simplifications to the elementary components, and compact forms of advanced components.

17.3.1 Bistable element

The most fundamental bistable element, the raw material from wuhich all flip flops are constructed, is a pair of inverters connected to each other. No matter whether the input to one of them is TRUE or FALSE, the circuit is always self consistent; yet the design is completely symmetric between the two alternatives.

17.3.2 Flip flop

To make an operable circuit, a bistable element must be provided with output leads

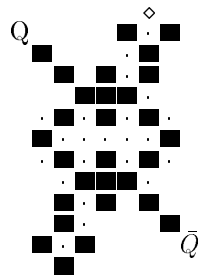

Figure 28: BISTABLE element

so that its state can be exploited by the remainder of the circuit. Additionally, there must be some mechanism to force it into a known state, which includes inducing it to change states. Rather than making the designer keep track of the state of the flip flop, the most elegant versions do this internally, automatically exchanging states upon the receipt of a triggering signal.

The following design incorporates a NOR gate; a single electron (arriving in the correct phase of the six phase clock), acting as a TRUE pulse, will force one or the other state, according to its entry point. The flip flop actually requires the TRUE signal for two cycles, which is arranged internally by pulse doublers.

An additional detail, not shown, would be to restrict the output stream to a single pulse. In that form the units could be cascaded to form a counter, the style of the output then being consistent with the input.

Starting from the indications shown here, any standard textbook on circuit design can be used to build up much more elaborate circuits.

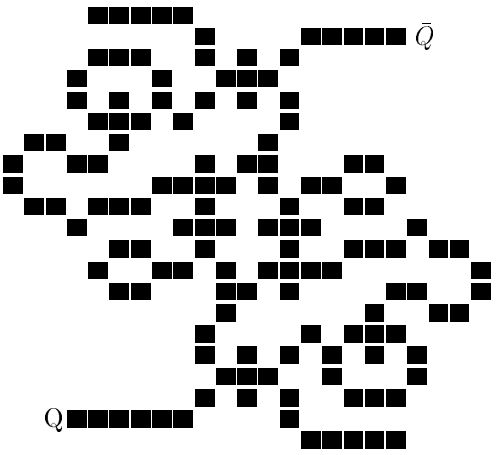

Figure 29: NOR gated FLIP FLOP

17.4 Advanced projects

Whether or not there is any point in using *WireWorld* to build up operating circuitry, there is no denying that it is very instructional, nor that the same expertise can be shared with the design of circuits in other media. Some differences exist, of course. *WireWorld* prefers NOR gates, whereas integrated circuit design is based upon NAND gates, for example.

17.4.1 Binary counter

The next step beyond a flipflop would seem to be a binary counter, obtained by connecting a string of flipflops, each element emitting a single signal as output for every two which it receives as input.

In order to function as a true counter, not simply a divider, the state of each stage always has to be available, as an output signal. But that is not a difficult detail.

17.4.2 Incrementer

Another basic digital circuit is the binary adder. Two input streams correspond to the summands; one wants an output stream representing their binary sum. Strictly, that is a serial adder; but a parallel adder would be mostly combinatorial.

An incrementer is a simpler variant; the output stream consists of an input stream to which a binary 1 has been added; connected to a loop, the circuit could count indefinitely.

17.4.3 Automaton

As an ultimate element of whimsey, one might emulate the CAM/PC, whose model is supposedly a long feedback loop with combinatorial logic for regenerating the loop using to the rule of evolution of the automaton.

17.5 Editing *WireWorld*

Unlike automata for which random fields suffice as initial conditions, and in common with *Life*, *WireWorld* requires a very carefully planned initial configuration. Loading previously prepared bitplane files is one solution to the problem, which merely places the responsibility for preparing the bitplane elsewhere.

With some ingenuity the Moore plane editor can be used to create a *WireWorld* configuration, modifying the planes one by one; but even better, by using the option `l` (edit planes) in CAMEX's main menu after the *WireWorld* option has been selected leads to a variant adapted to the needs of *WireWorld*.

The adaptation consists in allowing the symbols `2` and `3` to deposit electron states, and the suppression of all the options concerned with marking, modifying or displaying the rule table. Trial evolution, confined to the console screen, is still possible, as are the symmetry operations.

The generation of both random fields and random tables has also been omitted, just as the loading of alternative rules. However, INSERT *must* be used to load *WireWorld*; although a source of many accidents, that choice is consistent with universal browsing. Occasionally, the editing facilities may be desired for another rule, which ought not to be lost just because of temporarily entering the editor.

Since *WireWorld* requires three planes (two of them mostly empty), the *WireWorld* option `l` has its own private REC program, accessible via the usual `f3`, to store all three planes on disk. Standard names have been chosen for these files, but the REC editor can change them. It is recommended that the numerals 1, 2, and 4 always be incorporated in their names to facilitate remembering which arguments of `y` will be needed to load them again.

Option `f1` within the plane editor will exhibit a list of up to twenty files on the current disk with the extension `.PAT`, but INSERT will not work properly. Rather, the three planes must be loaded separately using option `y`; since `yp` can load multiple planes, correct arguments would be $p = 1, 2, 4$. As stated above, the number could be part of the file name, as in `WW4.PAT`.

The disk space required by most simple *WireWorld* demonstrations can be reduced by saving only the wire plane. Electrons can be inserted with the editor; if it is not obvious where they should go, pointers can be inserted among the wires as reminders. Good practice includes placing just enough extra lead at the inputs and outputs of sample components so that occurrence of electron heads and tails at their tips will coincide with transit times and internal clocks.

18 De Bruijn diagrams

The statistical properties of a cellular automaton can be predicted by drawing a "mean field curve" based on randomly distributing the cells throughout a configuration. Using the rules for compound probabilities in the evolutionary function predicts the distribution of the second generation, from which properties of the automaton, such as fixed points, can be inferred.

This approach must be corrected for the correlations which build up from generation to generation, creating a need for calculations which reveal the periodic configurations exactly; but then it is just as easy to include shifts along with the periodicity.

18.1 Neighborhood dominoes

Suppose that the neighborhoods of an automaton were split into several parts (typically, overlapping) which were used to construct tiles - dominoes if the sequence were linear. The fundamental problem is tiling the configuration space, ensuring that the states of the cells always coincide wherever the tiles overlap; a triviality in one dimension, but not always soluble for higher dimensions.

Suppose further that some of the tiles were withdrawn from the game; for instance those forming neighborhoods whose evolution did not proceed as desired, maybe tiles whose central cell changed between generations. Can large designs be constructed from the remaining tiles? And if so, how can they be described?

Evidently this procedure reveals configurations having periodic evolution, or consisting in periodic displacement. The longer the period, the larger the neighborhood required; each one requires enough internal information to guide its own evolution the necessary number of steps.

To avoid the possibility of an undecidable tiling in two or more dimensions, the search can be confined to configurations of a fixed spatial periodicity in all but the last dimension. The first step is to catalogue the tiles which can be strung out along a single direction, noting all the pairs which can be joined together.

18.2 De Bruijn matrix

There are (directed) graphs, usually called de Bruijn diagrams (for k symbols and n stages), whose nodes correspond to all the k^{n-1} possible sequences of length $n-1$ composed of the given symbols. Their links are sequences of length n, joining any pair of nodes where the tail comprises the first $n-1$ symbols, the head the last $n-1$ symbols.

In other words, 1101 would link node 110 to node 101 in a four stage binary diagram.

For two-dimensional binary automata with Moore neighborhoods, it is convenient to split the 3x3 neighborhood into two 3x2 rectangles, overlapping in a central column containing three cells. Accordingly its three stage, eight symbol de Bruin diagram would have sixty four nodes, each with eight links. Taking a full column as one single eight state cell makes the structure one dimensional.

Graphs are conveniently represented by matrices, whose rows and columns are labelled by the nodes. The elements of the matrix are to be ones and zeroes, according to whether the node indexing the row is linked to the node indexing the column; in symbolic form:

$$M_{ij} \quad = \quad \mathsf{link}(i,j).$$

Written with the link predicate, the rule of

matrix multiplication,

$$M^2_{ij} \quad = \quad \sum_k \mathsf{link}(i,k)\mathsf{link}(k,j),$$

implies that powers of such a matrix describe paths through the graph; the k^{th} power yields paths of length k, the sum of the first k powers paths of length k or less. Such matrix elements will be integers; those greater than 1 indicate a multiple path.

To make the general de Bruijn diagram correspond to a particular type of evolution, retain only those links whose neighborhoods behave properly. In the connection matrix, zeroes replace the missing links.

Powers of the de Bruijn matrix, the connectivity matrix of the diagram, reveal sequences of neighborhoods - rectangular regions - whose evolution proceeds correctly. In order to close the sequence into a ring of length k, the k^{th} power of the de Bruijn matrix should be calculated, from which paths corresponding to the diagonal should be culled.

18.3 Second level diagrams

Once acceptable strips extending along one dimension have been found, they can be stacked in some orthogonal direction - along a second dimension. The same procedure applies as before; each ring is split into two overlapping subrings, following which a diagram (or its connectivity matrix) is prepared showing how to combine them into longer entities.

Here there is an overt problem which is only implicit in the first level diagram, namely that links ought to be eliminated which terminate in dead ends; since they cannot participate in arbitrarily large configurations. Although dead ends occur at the first level, they get discarded whenever closed loops are extracted from the diagram.

At the second, or final level, it is likely that only a finite representative of the infinite plane will be exhibited, so it is necessary to know in advance if some path will eventually be blocked. The solution is to program a recursive scan of the proposed second level diagram, in which all nodes are eliminated which have no incoming arrows, as well as those which have no outgoing arrows.

The final product will contain only loops joined to each other. There is no guarantee that there will be global loops; in *Life* this lack produces the phenomenon known as "fuses." It is possible to circulate in one part of a graph for an arbitrarily long time, later crossing over to another part, from which there is no return; but where there are new loops in which to continue circulating.

Of course this can happen several times, producing composite fuses. In the terminology of matrix theory, such connectivity matrices have block triangular form.

18.4 Third level diagrams

For automata with more than two dimensions, higher dimensional slabs can be built up dimension by dimension by repeating the process already described. The volume of the Moore neighborhoods involved increases rapidly with dimension, limiting the practical feasibility of going beyond two dimensions for most automata.

18.5 The de Bruijn option

The main CAMEX menu contains an option `d` for generating de Bruijn diagrams for Moore automata. The rule table has to bave been selected previously; option `d` uses whatever table is currently resident in the array `mogrul` (which has not necessarily yet been loaded into the CAM lookup table).

Within the option, a graphics mode screen is shown, containing three main panels and space for several headers and items of data.

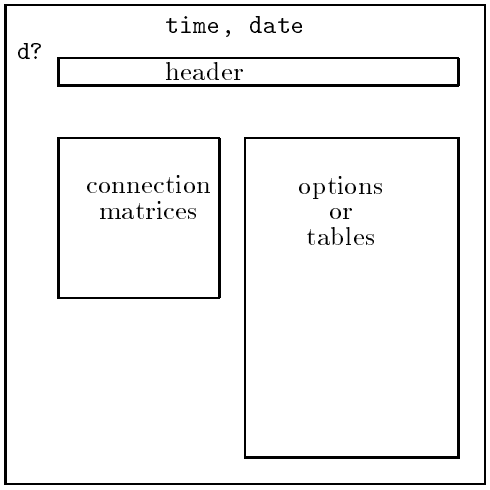

Figure 30: de Bruijn panels

The submenu which is displayed admits the following options; all but the obvious ones are later explained individually:

- abcdef - shifts,
- ABCDEF - strips,
- MNOPQR - 0, 0's,
- 123456 - sample,
- SsGg - CAM, console,
- = - type matrix,
- i - matrix grid,
- Kk - disk output,
- mno - graph matrix,
- t - matrix power,
- 8 - survey 1-9,
- /, z - clear screen,
- Z - clear all,
- <cr> - exit,

18.5.1 Shifts

There are several directions in which a shift can take place, not to mention the possibility of none at all. Evolution into a constant state is also envisioned.

a - still life; no shift.
b - transversal; shift one cell across the strip.
c - diagonal; shift one cell along, one across the strip.
d - longitudinal; shift one cell along the strip.
e - vanishing; every cell enters state 0.
f - setting; every cell enters state 1.

The shift option clears the matrix panel, then displays a 64x64 matrix of 1x1 pixels showing which of the partial neighborhoods overlap consistently. The outline of a three stage, eight symbol de Bruijn matrix is always in evidence.

18.5.2 Periodic strips

The width of the strip chosen for the first level de Bruijn diagram varies from 1 to 6 according to the options A to F. Wider strips are not available simply because of memory limitations, but execution time would also be a consideration if the strips were much wider.

A	1	B	2	C	3
D	4	E	5	F	6

The strip option calculates the second level de Bruijn diagram, which describes the sequencing of the partial strips (actually, it has been edited so that only loops remain). The number of nodes can be very large - up to $n = 2^{2w}$ for a strip of width w - , so the final display is adjusted accordingly. If $m \leq 64$, the connectivity matrix is displayed using 2x2 pixels on a green background.

Otherwise, if $m \leq 128$, the connectivity matrix is displayed with 1x1 pixels. If there are still more nodes, no matrix at all is shown.

In every case, it is possible to obtain a listing of the matrix elements of the second level matrix.

18.5.3 Isolated strips

Slightly wider strips can be accommodated within the available computational space if they are supposed to be isolated - that is, beginning and ending with at least two cells in state 0 (supposedly quiescent) - for neighborhoods of radius 1 such as CAMEX uses. If complete isolation is desired, each strip produced by the second stage has to be checked for eventual zero termination for both orientations along the cross direction.

M	1	N	2	O	3
P	4	Q	5	R	6

The same presentation which is given to the periodic strips is given to the isolated strips; however, they are usually far less in number, making wider strips feasible.

18.5.4 Sample

For those cases where the resulting number of nodes was small enough to record the second level connectivity matrix, sample configurations can be generated, both on the console screen and in CAM bitplane 0. A random number generator is used to read the graph, taking each branch with equal probability. Consequently a different pattern will usually be generated each time this option is used.

1	1	2	2	3	3
4	4	5	5	6	6

18.5.5 To run the sample

Following the overall CAMEX conventions, `S` and `s` will set evolution going in the CAM, whilst `G` and `g` apply to the console screen, where the limitations of small size and non-cyclic borders apply.

As they evolve, even the sample CAM screens will degenerate, for two reasons: Unless the width of the strip is a divisor of 256, the width of the CAM screen (or of the width of the console screen, as appropriate), the configuration will not be truly periodic; only in rare instances will this be immaterial.

The screen is generated from the center row upwards and downwards; there is no guarantee that joining will occur in the vertical direction either. Even if the second level de Bruijn diagram permitted it, the program would make no effort to ensure vertical closure. So all the allowable configurations can materialize, but only as finite extracts from the infinite plane.

18.5.6 Disk output

The connectivity matrix of the second level de Bruin diagram is often rather large, usually exceeding the capacity of the screen display. Most of the time it is not required, but occasionally one wants to see it; even when it is small there are times when a permanent record is desirable. The option `k` enables disk output, `K` suppresses it.

No information is actually written on disk by `k` or `K`; they decide whether one of the options that generate strips, periodic or isolated, will write the links on file `P0.DAT`. Furthermore, the execution of one of those options (`A ... F, M ... R`) is the only occasion on which the file can be written.

The file name is fixed; later files will overate earlier ones. To save several files requires

executing CAMEX several times, renaming the file every time.

Once produced, a file can be sorted with the DOS utility SORT. Sorted according to either incoming or outgoing links, the lists tend to be more readable than in their natural form, which follows chains as far as possible before returning to branches encountered earlier.

18.5.7 Matrix power

Once the connectivity matrix for the first level de Bruijn diagram has been set up, it can be raised to powers; each time option `t` is used, the power is increased by 1. At the same time, the power matrix is displayed in the matrix panel; from there it can be analyzed, even copied by a screen dump if it is so desired.

By exhibiting a visual image of the matrix, some of the details of the linkages can be seen, such as their number and density. The matrix is not sorted, so all but the most obvious block diagonal or block triangular forms tend to be obscured until the higher powers.

The matrix display distinguishes only nonzero elements, but the $(0,0)$ element and the trace are always shown just above the header line; 0 is always the number of the zero partial neighborhood.

Two matrices are maintained by the program. One is the first level connection matrix, displayed by typing `m`. The other is a work matrix; it contains the powers generated by option `t`, and can be displayed by typing `n`.

When the work matrix is used to hold the second level diagram, the labelling arrangement is different; the matrix uses a coded form of the link as the link's matrix element, `-1` for a non-link. The matrix can be viewed using option `o` instead of `n`.

Conversion to standard form can be effected by option `~`, but then no more samples can be generated without starting all over again. On the other hand, `t` and option `8` could then be used to gather statistics on the second level diagram.

18.5.8 General survey

If it is only required to know how many horizontal strips of a given length there are, but not their exact composition, option `8` may be used. The de Bruijn matrix and its first eight powers are calculated; their $(0,0)$ elements and traces are tabulated.

The $(0,0)$ elements tell how many isolated strips there are, the trace the number of periodic strips. Not all these strips will stack vertically, so the numbers represent an upper bound (but often a fair estimate) for the number of nodes in the second level diagram. The maximum number of links each node may have is 2^{l-2} (isolated) or 2^l (periodic), for length l; in practice the number is much, much lower.

18.6 Typical operation

To calculate a de Bruijn diagram, and particularly, to set up some examples whose evolution can be checked by the CAM, the following steps could be followed.

First, select the rule, and be sure that it is installed both in `mogrul` and the CAM lookup tables. It is better to do this explicitly rather than relying on one of the REC demonstrations to leave the tables behind. The INSERT option within the `f1` options which allow rule editing will usually load the tables correctly.

The main menu option `d` affords the only access to de Moore $(2,1)$ de Bruijn diagrams (access to Moore $(4,1/2)$ diagrams resides in the menu for that option, but it is much

more limited and requires far less space); the scarcity of data space for CAMEX precludes having any additional data structures present.

Within the de Bruijn submenu, the class of shift should be selected at once; otherwise the connectivity matrix will be uninitialized, making further results meaningless.

The general survey, matrix powers, or a second level diagram can now be selected.

Having generated a second level matrix, it can be saved on the disk, or used to generate samples on the screens. Only one matrix can be saved per session, the latest if there were several attempts. Samples can be generated and run indefinitely.

Screen samples should confirm whatever type of evolution was selected, but the inevitable discontinuity at the edges of the screen will usually erode the image, starting at the lines where the periodicity fails.

It is possible to begin over again at any time by going back to select a shift class. Different widths within the same class can also be chosen, without having to go all the way back to the shift class options.

The rule table remains intact when returning to the main menu, but all the connection matrices will have to be reestablished when reentering `d` later on.

Whether the rule has changed in the interim depends on whether `mogrul` has been altered during that time; von Neumann options leave it intact, but having used INSERT during any Moore option will have surely resulted in change.

18.7 External data bases

The use of de Bruijn diagrams is somewhat varied. Theoretically, they establish the existence of many phenomena and limits which otherwise could have gone unrecognized. In that respect they are very useful, even when they are never calculated at all; but once a calculation is undertaken, it usually turns out to be very lengthy.

There are two instances in which de Bruijn diagrams appear within CAMEX, and a third where it could be included were the demand to materialize.

The first, option `d` in the main menu, applies to $(2,1)$ Moore automata in two dimensions, having been discussed here at some length. Only first generation diagrams were possible; their inclusion required substantial revision of the layout of CAMEX's data segment. Even so, the strips that can be gotten must be less than six cells wide.

The second, option `j` within the $(4,1/2)$ Moore automata, is even more restricted; only first generation diagrams with strips of width less that three cells can be realized.

A third possibility would be to annex LCAU's de Bruijn modules to CAMEX, but the prospective user would be well advised to go directly to the corresponding LCAU program, where there is intrinsically a greater variety of automata to choose from. Even so, the `64K` barrier still exists; somewhat longer strips still fit the limitation.

Life is one of the few automata for which a significant demand has actually arisen for extensive results of de Bruijn calculations; this is no doubt due to *Life*'s high recreational value.

Considering that the final diagram is fairly modest, particularly having taken into account the effort required to obtain it, suggests the formation of a data base from which *Life* patterns can be generated on demand.

The important parameter is the size of the final diagram, rather than the demands upon the program which created it. Efforts are underway to create this data; as it becomes available the final diagrams will be adjoined

to CAMEX.

19 CAM hardware

Although the circuit diagrams for all the different CAMs seem to be proprietary information, published descriptions allow working up at least a block diagram of their operation; contemplating software which interacts directly with the CAM without such information would appear to be rather difficult.

One of the important characteristics of a CAM is its role as a video controller, as a consequence of which it can be implemented as a specialized shift register. Rather than building a fully parallel circuit in which every cell evolves simultaneously, it is only necessary that each cell be updated as its place on the monitor screen comes around. The process evidently favors two-dimensional cellular automata.

Three essential components of a CAM have to be considered: output to the video monitor, data defining the states of the cells, and the rule of evolution of the automaton. Choice of these parameters is restricted and influenced by permanent elements of the circuit design, such as how many states are allocated to each cell, and the neighborhood which will surround it. To a certain extent, these details are also programmable.

The fundamental structural element of a CAM is a bitplane; in the CAM/PC it is a 256x256 binary array. Grouping two bitplanes into a half-CAM gives a full CAM a complement of four bitplanes. The peculiar nomenclature results from an assymetry in the two planes of a half-CAM — sometimes only one of them evolves. Following mathematical and computing convention, the four planes are numbered 0, 1, 2, 3.

The way that a linear shift register of 64K stages can be treated as a plane is to take out taps at 256 bit intervals; it is sufficient to read from the feedback point of the cycle, 1 and 256 bits ahead, also 1 and 256 bits behind to obtain a five cell, two dimensional, binary von Neumann neighborhood.

20 Modifications and extensions

20.1 Hardware origin

20.2 Radius 1/2 Moore neighborhood

21 Acknowledgements

The CAM/PC board which has been used for the development of CAMEX was purchased with a grant from Mexico's *Consejo Nacional de Ciencia y Tecnología* (CONACYT). We are grateful to *Automatrix, Inc.* for selection as a beta test site, and for consultation on various aspects of the board's programming and performance.

References

[1] Ivan Amato, "Speculating in Precious Computronium," *Science* **253** 856-857 (1991).

[2] Bernard Barral, Hugues Chaté and Paul Manneville, "Collective behaviors in a family of high-dimensional automata," *Physics Letters A* **163** 279-285 (1992).

[3] P. -M. Binder and V. Privman, "Second-Order Dynamics in the Collective Temporal Evolution of Complex Systems," *Physical Review Letters 68* 3830-3833 (1992).

[4] H. Chaté and P. Manneville, "Evidence of Collective Behaviour in Cellular Automata," *Europhysics Letters* **14** 409-413 (1991).

[5] A. K. Dewdney, "The hodgepodge machine makes waves," *Scientific American*, pages 86-89 (August, 1988).

[6] A. K. Dewdney, "Computer Recreations: The cellular automata programs that create wireworld, rugworld, and other diversions," *Scientific American*, pages 136-139 (January, 1990).

[7] Richard Durrett and Jeffrey E. Steif, "Some Rigorous Results for the Greenberg-Hastings Model," *Journal of Theoretical Probability* **4** 669-690 (1991).

[8] Irving R. Epstein, "Spiral Waves in Chemistry and Biology," *Science* **252** cover + p. 67 (1991).

[9] Marcel J. E. Golay, "Hexagonal Parallel Pattern Transformations," *IEEE Transactions on Computers* **C-18** 733-740 (1969).

[10] J. M. Greenberg and S. P. Hastings, "Spatial patterns for discrete models of diffusion in excitable media," *SIAM Journal on Applied Mathematics* **34** 515-523 (1978).

[11] J. M. Greenberg, B. D. Hassard, and S. P. Hastings, "Pattern formation and periodic structures in systems modelled by reaction-diffusion equations," *Bulletin of the American Mathematical Society* **84** 1296-1327 (1978).

[12] J. M. Greenberg, C. Greene, and S. Hastings, "A combinatorial problem arising in the study of reaction-diffusion equations," *SIAM Journal of Algebra and Discrete Mathematics* **1** 34-42 (1980).

[13] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1978 (ISBN 0-13-110163-3).

[14] Don Lancaster, "TTL Cookbook," Howard W. Sams & Company, Indianapolis, Indiana, 1974 (ISBN 0-672-21035-5).

[15] Harold V. McIntosh and Gerardo Cisneros, "The programming languages REC and Convert," *SIGPLAN Notices*, **25** 81-94 (July 1990).

[16] Harold V. McIntosh, *Linear Cellular Automata*, classroom notes, Universidad Autónoma de Puebla, 1990 (194 pp).

[17] James D. Murray, "How the Leopard Gets its Spots," *Scientific American*, pages 62-69 (March, 1988).

[18] William Poundstone, *The Recursive Universe*, William Morrow and Company, New York, 1985 (ISBN 0-688-03975-8).

[19] Kendall Preston, Jr., "Feature Extraction by Golay Hexagonal Pattern Transforms," *IEEE Transactions on Computers* **C-20** 1007-1014 (1971).

[20] Kendall Preston, Jr. and Michael J. B. Duff, *Modern Cellular Automata*, Plenum Press, New York, 1984 (ISBN 0-306-41737-5).

[21] Peter R. Rony, "Introductory Experiments in Digital Electronics and 8080A Microcomputer Programming and Interfacing - Book 1," Howard W. Sams & Company, Indianapolis, Indiana, 1977 (ISBN 0-672-21550-0).

[22] Tommaso Toffoli and Norman Margolis, *Cellular Automata Machines*, The MIT Press, Cambridge, Massachussets, 1987. (ISBN 0-262-20060-0)

[23] Arthur T. Winfree, "Rotating Chemical Reactions," *Scientific American*, pages 82-95 (June, 1974).

[24] A. T. Winfree, E. M. Winfree and H. Seifert, "Organizing centers in a cellular excitable medium," *Physica* **17D** 109-115 (1985).