VideoEasel

Thomas Richter

COLLABORATORS			
	TITLE : VideoEasel		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Thomas Richter	April 14, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 VideoEasel **1**

1.1 VideoEasel Guide . . . 1
1.2 The THOR-Software Licence . . . 3
1.3 My address and EMail . . . 4
1.4 What VideoEasel is used for . . . 4
1.5 Requirements . . . 4
1.6 Installing VideoEasel . . . 4
1.7 What about it ? . . . 5
1.8 About the name and the history of this program . . . 7
1.9 thank you goes to.... . . . 7
1.10 The experiments section . . . 8
1.11 Experiments for Fun . . . 9
1.12 Scientific Experiments . . . 9
1.13 A live demo . . . 10
1.14 Going into the reverse gear . . . 13
1.15 TRON choreography . . . 15
1.16 Critters swarms . . . 15
1.17 Cellular computing . . . 16

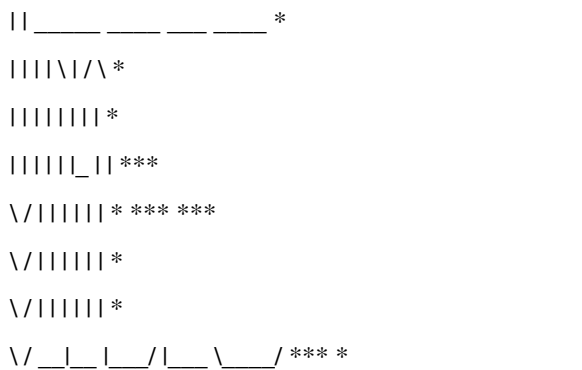
Chapter 1

VideoEasel

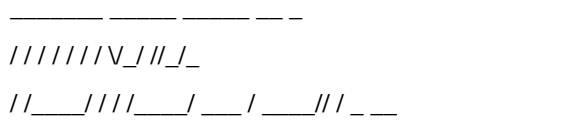
1.1 VideoEasel Guide

VideoEasel Guide

Demo Guide Version 1.00 VideoEasel Version 1.60



by



/ / / / / / \ / /\/ / / / / /_/ / /_/_/

/ / / /____/ / / _/__/_/ /__/_/_/_/ /_/_/ /_

/

Please note:

This is NOT the full VideoEasel guide! It's only a demonstration

about what will be included in the full version.

To order the full version, send $20 to my address .

Table of Contents

I. The Licence

Legal stuff!

II. What is it: Overview

What it does...

III. Requirements

You need to provide...

IV. Installation

How to install VideoEasel.

V. Introduction

What are cellular automata, then?

VI. Using VideoEasel for drawing

Draw pictures using VideoEasel, setup the

start configuration for the automata.

VII. Experiments using VideoEasel

For fun and for investigations.

VIII. Programming VideoEasel

How to write own automata, applications, algorithms, video drivers and more!

Useful refrences:

1. The VideoEasel Menu

The complete menu list as a short refrer

2. The VideoEasel Slang

Often used words.

3. About VideoEasel

The history of this program.

© THOR-Software

Thomas Richter

Rühmkorffstraße 10A

12209 Berlin

Germany

EMail: thor@einstein.math.tu-berlin.de

WWW: http://www.math.tu-berlin.de/~thor/thor/index.html

1.2 The THOR-Software Licence

The THOR-Software Licence (v2, 24th June 1998)

This License applies to the computer programs known as "VideoEasel" and the "VideoEasel_Demo.guide". The "Program", below, refers to such program. The "Archive" refers to the package of distribution, as prepared by the author of the Program, Thomas Richter. Each licensee is addressed as "you".

The Program and the data in the archive are freely distributable under the restrictions stated below, but are also Copyright (c) Thomas Richter.

Distribution of the Program, the Archive and the data in the Archive by a commercial organization without written permission from the author to any third party is prohibited if any payment is made in connection with such distribution, whether directly (as in payment for a copy of the Program) or indirectly (as in payment for some service related to the Program, or payment for some product or service that includes a copy of the Program "without charge"; these are only examples, and not an exhaustive enumeration of prohibited activities).

However, the following methods of distribution involving payment shall not in and of themselves be a violation of this restriction:

(i) Posting the Program on a public access information storage and retrieval service for which a fee is received for retrieving information (such as an on-line service), provided that the fee is not content-dependent (i.e., the fee would be the same for retrieving the same volume of information consisting of random data).

(ii) Distributing the Program on a CD-ROM, provided that

a) the Archive is reproduced entirely and verbatim on such CD-ROM, including especially this licence agreement;

b) the CD-ROM is made available to the public for a nominal fee only,

c) a copy of the CD is made available to the author for free except for shipment costs, and

d) provided further that all information on such CD-ROM is redistributable for non-commercial purposes without charge.

Redistribution of a modified version of the Archive, the Program or the contents of the Archive is prohibited in any way, by any organization, regardless whether commercial or non-commercial. Everything must be kept together, in original and unmodified form.

Limitations.

THE PROGRAM IS PROVIDED TO YOU "AS IS", WITHOUT WARRANTY. THERE IS NO WARRANTY FOR THE PROGRAM, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IF YOU DO NOT ACCEPT THIS LICENCE, YOU MUST DELETE THE PROGRAM, THE ARCHIVE AND ALL DATA OF THIS ARCHIVE FROM YOUR STORAGE SYSTEM. YOU ACCEPT THIS LICENCE BY USING OR REDISTRIBUTING THE PROGRAM.

Thomas Richter

1.3 My address and EMail

My Address:

Thomas Richter

Rühmkorffstraße 10A

12209 Berlin

Germany

EMail: thor@einstein.math.tu-berlin.de

WWW: http://www.math.tu-berlin.de/~thor/thor/index.html

1.4 What VideoEasel is used for

VideoEasel is a generic environment for so called cellular automata machines. The most popular amongst them is Conway's "Life"-algorithm, which has been implemented on almost every machine. But VideoEasel does much more: You may modify the rules of the game, may implement easely other algorithms, modify existing ones... I implemented every single CAM (Cellular automata machine) I could find... That was much work, and for that reason this guide is copyrighted. Read the licence!

Continue here for a introduction .

1.5 Requirements

VideoEasel is in fact very demanding. 1MB chip mem is a must, 2MB would be fine. Fast memory MUST be provided, the more, the better. I think the minimum should be somewhere around 3MB (Argh!). The point is that I wanted to make this monster fast and I have enough (10MB) memory. Processor: Well a MC68000 works, but... As minimum, I would suggest a MC68020 with 25 MHz, the more, the better....

1.6 Installing VideoEasel

Please download the VideoEasel archive first, if not already done. It can be found in the AmiNet, or at my home page, see here .

Download the "RexxMathLib" archive as well. Again, this is on my home page or in the net.

After extracting the archive, copy the extracted files whereever you like.

Extract now the "RexxMathLib" archive and follow the instructions given there to install it (If I remember exactly, you only have to copy the library to LIBS:...).

Copy the "PatchDraw" program to your C: directory and add the following line to "S:Startup-Sequence" using a standard editor:

SetPatch >NIL:

PatchDraw ; <-- Insert this here...

; more stuff

"PatchDraw" corrects a flaw in the graphics.library line drawing algorithm,

it does no specific VideoEasel patch.

If you lost the PatchDraw command, contact me to receive a copy.

1.7 What about it ?

So what is a cellular automata then?

Think of the computer screen as devided into tiny squares like the squares on checkered paper. Each tiny square is called a "cell" of the automata, and each cell can be filled with one of several defined "colors". The coloring of the square is called "the state" of "the cell".

Now a "dynamics" is introduced on the cells, i.e. we define rules how to change colors of the squares depending on the colors of the "neighbour" cells. The term "neighbour" is quite flexible and can be defined with VideoEasel, but let us introduce two popular definitions for which cells are "near" to a cell: The cells marked with a "+" sign are "near" to the "*" cell in this drawing:

Von Neumann-neighborhood: Moore-neighborhood:

+ +++

+*+ +*+

+ +++

The next "state" (color) of the cell "*" depends now on the state (colors) of the "+" cells, as marked in the drawing above.

Sounds boring? Well, maybe. But let's start with a simple example: We choose two possible states for a cell, say black and white. We choose the Moore-neighborhood as the neighbors of our cell. Take a piece of checkered paper, and draw the following initial conditions of the "automata", i.e. fill three adjacent squares like the following:

***

We have to define the "dynamics" of the automata. The rule goes like the following:

Take a second, empty, sheet of paper, and count the (Moore-)neighbors of each cell. If the original cell is black itself and has two or three black neighbors, paint it black on the second sheet. If the original cell is white and has exactly three black neighbors, paint it black on the second sheet as well. Leave all other cells on the second sheet empty, i.e. white. By applying this rule to the configuration above we'll see that:

The two cells at the side turn into white on the second sheet as both of them have only one black neighbor.

The cell in the center stays black since it has two neighors.

The cells on top and below of the cell in the center turn black as well since both have three neighbors on the first page: Remember - neighor-counting is done on the first page, coloring on the second.

So we get, after "one iteration" of the automata the following configuration:

*

*

*

The bar has turned by 90 degrees. Applying the same rule again on a third sheet reveals the original bar, so this configuration flips forth and back each iteration of the automata. It is called for this reason a "blinker".

Still boring, ey? O.K. let's start a more complex example, we begin with the following arrow-like configuration:

***

*

*

After applying the rule above, we get:

*

**

* *

and again:

**

* *

*

and again:

**

**

*

and again:

***

*

*

Which is again your start configuration. However, if you did your work carefully, you should notice that the whole configuration has moved one square to the left and one square to the top, so we have here a configuration that travels diagonal thru space. It's called for this reason a "glider". Quite interesting things could happen if a "glider" crashes into a blinker: The glider gets reflected, or a block is created and the glider vanishes:

**

** this is a "block". It is "stable" and does not

change with each iteration.

So we've here a playfield with objects of their own dynamics, without having a rule that says: "Each glider must be moved one pixel in a diagonal direction each step". It does this by itself, just by the simple rules we constructed. In fact, quite a lot of machines can be constructed: A glider "factory" that produces gliders, as well as configurations that move horizontal and vertical, so called "fishes", and "factories" for fishes, even a complete "computer" can be constructed just by this simple rule!

Each electrical part in a computer can be build on top of this machinery, with gliders and fishes acting as a replacement for the electrical signals in a usual computer. It's in fact a whole universe of its own!

If you want a quick test of this automata: It's impletented in VideoEasel, under the name of "Life" which was choosen by its inventor, J. H. Conway.

-Start VideoEasel.

-Select the menu point "Open" in the "Apps" menu. A file requester should pop up.

-Locate the file called "Life", select it and press "Open".

VideoEasel will ask you if it's O.K. to clear the screen. Answer "O.K.".

-Select the blue color for drawing by moving the mouse over the blue rectangle at the lower right bottom of the screen and pressing the left mouse button.

-Select "Load" from the "Brush Menu". A requester should appear.

-Select the file "glidergun.life" and press "Load".

A messy shape will be attached to your mouse pointer now. Move the mouse pointer somewhere in the middle of the screen and press the left mouse button once. This will place this obstacle in the screen. The color blue is now similar to "black" and the dark background on the screen was "white" in the last do-it-yourself example.

-Start the automata by pressing the "|>" shaped button right on the screen and see what happens: This will start the calculation of new "generations" of this automata, exactly like you did it before with pen and paper.

The "obstacle" will begin to blow up and shrink periodically and spits out a glider each several cycles of the automata. The gliders will start moving in the lower right direction.

-Stop the automata with the same button on the tool bar right of the drawing area (it looks now like a filled circle). Load another brush with the same menu and place it somewhere on the screen, try to experiment!

Nice game, isn't it? So what is this good for, at least (except having some fun)?

There are some automata with a bit more "serious" applications than what we've seen before with "Life". Here's just one of them:

-Select again "Open" from the "Application" menu like before. Select the "CAMDendrite" and press "Open".

-Select "Random Fill" from the picture menu. This will fill the complete screen with randomly selected colors.

-Select the light white color on top of the palette with the left mouse button as before.

-Move the mouse pointer to somewhat the middle of the screen and press the left mouse button once. This will give a single white dot.

-Select "Speed" from the "Apps" menu. Move the slider all to the left for maximal speed. The number below should become a zero. Press "O.K.".

-Now start the automata as before and watch carefully.

You'll see that colored particles move around the screen, and once they touch the particle on the middle, they stick to it and stop moving. The whole configuration in the middle starts growing in all directions and will show a "dendrite" like structure.

Here we have an automata that somehow "models" the growth of a dendrite in nature. Again, there's no rule that a dendrite should look like this, but this simplified rule of the automata (which is too complicated to be explained here) models the more complex rules in nature, resulting in the same sort of structure comming out.

Much more physical laws than the dendrite growth can be modelled with cellular automata, for example ferromagnetism (the "Ising model") and the gas laws are another application. Even the propagation of sound in gas can be simulated, as well as the focussing of sound by an optical "mirror". Much more on this in the Experimental section of the guide.

1.8 About the name and the history of this program

You might ask why this program is called "Video Easel". To be honest, I don't know, at least not really.... (-:

For sure, it's an electronic Easel for generating nice "Video" like movies, but the real reason is different:

Another program like this one was written years before, not running on the Amiga, but on an old antique eight bit Atari XL. Only one automata was present, the popular "Life" I described before. Drawing with this program was a horror, but it was quite fast for its time. Guess what: It was called "VideoEasel" as well.

Later on, I read several articles in the scientific american and got really curious about how this might work on my computer. A first version was written very quickly, in assembly language. It got never stable, so I quit this project...

Years later I started another attempt, this time in C and assembler. C for all control functions, assembly language for the fast part, like the automatas itself. The result facinates me so much that after the program was half done I couldn't stop playing with it for several weeks, delaying the whole project considerably. Then I found a book by Toffoli and Margolus (see again here for a reference), describing much more automatas. A lot of improvements were necessary to implement them all, but after two years of testing, trying and implementing, this is the result.

Hope you enjoy it as much as I do.

1.9 thank you goes to....

VideoEasel credits page:

Tomas Rokicki, for his FastLife algorithm. I haven't used it in

VideoEasel, but was inspirited by the idea. (Ain't there a 'h'

missing in your name, T(h)omas?)

Ron Charlton for his FastLife program. I used many patterns found

in his archive. They (and much more) are contained in the

XLife folder. The ready to use brushes are in the brush directory.
Jon Bennett for his XLife implemenation for Unix. Again, some ideas
have been adapted. All patterns found in the XLife archive are
included in VideoEasel, although the import script does not yet
handle the new include style yet...
Brian Hayes and A.K. Dewdney for their arcticles in the Scientific
American (Spektrum der Wissenschaft in german). They brought me to
the idea writing VideoEasel. Some algorithms are based on their
articles (The wire world, Life, Crystal and City...)
Tommaso Toffoli and Norman Margolus for their book on cellular
automata machines. All "CAM"-type algorithms are based on their
book and have been translated from CAMForth to CAMRexx.
Thanx does not go to...
Commodore/Amiga for their #$?&! computer. This sh*tty thing never
worked like it should (third computer, 9th ! mouse, 3rd fan, 2nd
board update...). I really LIKED it when the went bankrupt!
All the users that work with VideoEasel without buying the manual...

1.10 The experiments section

Here we're going to discuss a lot of tiny experiments that can be done with VideoEasel in case you got curious what this program is good for.

This chapter is divided in two sections, namely:

Experiments for Fun

A lot of experiments that produce at least some colorful and
nice graphics. This could be seen as a "good enough" reason
to play with them, and I haven't found any other application
for them.

Scientific experiments

Some "serious" applications follow. Don't worry, it's at
least as entertaining as the first section!
Some of the automata discussed here have been constructed
to model parts of "nature" in a computer, hence you might
discover quite interesting experiments here.

1.11 Experiments for Fun

None of the experiments below have a serious application - at least I can't think of any application. However, they might be quite funny to make...

The inkspot simulation

A first simple step.

Diamonds, triangles and squares

Plant diamonds and other germs.

The automatic duplicator

Copy shapes with this one.

Fractal carpets

Complex shapes with simple rules.

Fractal mazes

The fractal labyrinth generators.

A live demo

The most popular LIFE automata.

Brian's brain

Gliders forever in your brain.

Greenberg's rule

Quite different behaiviour with one automata.

Going into the reverse gear

Time reversal christmas stars.

An impenetrable shield

Jump thru the time with the Time Tunnel.

TRON choreography

A simple margolus game.

Critters swarms

Beasty flattering objects.

Cellular computing

What? A computer inside a computer ?

Wire world

More circuits in cellular automata.

Micro electronics in Digital logic

Simpler circuits in CAMRexx.

1.12 Scientific Experiments

This section covers some experiments with a more or less serious applications. Some of the models described below have been discussed by serious mathematicians and phycicists, however mostly without animating them on the computer.

1.13 A live demo

This automata was introduced 1970 by the mathematician John Conway. It caused attention of science amateurs and professionals all over the world. LIFE may be thought of as describing a population of stylized organisms, developing in time under the effect of counteraction propagation and extinction tendencies.

A first experiment:

- Open the "Life" application. There's more than one implementation of this automata, namely "FastLife", "FreeLife", "BWLife", "CAMLife", "CAMLife_Echo", "FastBWLife", "SmartLife" and "LightspeedLife". All of them use the same rules for creating the next generation, but with quite different approaches. "LightspeedLife" is the fastest, but only black and white. "FastLife" is more colorful, but slower and usually a good compromise between colorfulness and speed. "LightspeedLife" is very restricted in its parameters, the "FastLife" provides more freedom. If even more parameters are wanted, switch to "SmartLife". Its even slower than "FastLife", but more flexible. Most useful configurations have been developped for the simple "Life" automata, so let's use this one for the first try.

- Load the brush "gliderglidergun.life" and place it somewhere in the middle of the screen.

- Start the automata.

- Increase the speed if necessary.

What you'll see are twelve "gliders" colliding together, resulting in a machinery that spits out gliders periodically. The gliders will start moving to the lower edge of the screen and will crash into the boundary, causing some boundary artefacts.

How does it work?

Each cell has nine Moore neighbours and can be either turned on or off. An "off" cell is colored black, an "on" cell has one of three colors used to indicate if it has been turned on in the last step ("green") or if it survived the last step with two ("blue") or three ("pink") neighbours. The rules of "birth" and "death" of each cell go like follows:

- An "off" cell is turned on ("birth") if it has exactly three neighbours. It is colored "green".

- A cell already "on" will stay "on" if it has exactly two or three neighbours. Cells with two neighbours are colored blue, cells with three will be shown in pink in the next step.

- All other cells "die", i.e. will be shown in black.

More experiments:

Much more must be said about the LIFE automata. We start with some basic creatures in this "Life" universe. You've just discovered one of them, the "glider":

***

*

*

and its rotated forms. It moves with the fastest possible speed in this universe ("light speed") in a diagonal direction. Two other basic figures are:

The Blinker The Block

*** **

**

The blinker flips 90 degrees each step, and the block doesn't move at all.

Remember, these figures aren't coded into the automata, they just come out by applying the rules I gave above. To illustrate this, consider the "block" on the right hand side: Each cell in it has exactly three neighbours, hence it stays on forever. The block does not move and does not change either.

The blinker is a bit more complicated to handle: The cell in the middle is "on" and has three neighbours, hence stays on. The cells to the left and the right have too few neighbours to remain on, but the cells on top and below the middle cell have exactly three cells, so are turned on. Hence the whole object rotates in one step of the automata.

An even more complicated machinery was shown in the experiment above. This is the so called "glider gun" or "glider factory". As you've seen, this configuration moves and changes a bit, spits out a glider and restores itself some cycles later. The period is exactly 30 cycles of the automata.

It is said that this configuration was found by some students who have won a bet about wether it is possible to create an automata that grows infinitely.

If we forget about the boundary just for a moment, this glider gun creates one glider each 30 steps, hence grows by one glider each 30 steps. The gliders would last forever if they won't crash into the boundary, resulting in this infinite growth. However, it is possible to create configurations that grow even faster than this simple glider gun...

The next experiment introduces another basic configuration worth mentioning:

- Clear the screen.

- Load the brush "gunandeater.life" and place it somewhere in the middle of the screen.

- Start the automata.

You'll see the glider gun of the last experiment again, spitting out a glider each cycle. The glider itself crashes into a configuration called the "eater". The "eater" itself survives this crash without damage, but the glider is destroyed:

The Eater

**

*

***

*

This configuration has been proven to be astonishingly stable. Almost nothing can destroy an "eater".

We can try some "crash tests" by directing gliders into other obstacles:

- Clean the screen.

- Reload the "gunandeater.life" brush.

- Place the brush in the middle of the screen.

- Select dots for drawing by clicking at the upper left crosshair tool in the toolbar.

- Open the magnifier tool and place it on top of the "eater".

- Select the blue color.

- Click into the window.

- Add one point to the eater so that the upper part of it looks like a block:

**

*+ <- add this point.

*** <- remove this line

* <- remove this line as well

- Select the black color.

- Click into the window again and erase the lines below the block as shown as in the drawing above.

- Start the automata.

You'll see the glider gun generating one glider. This glider crashes into the block, removing both the block and the glider.

Hence, blocks can be anihilated by gliders crashing into them. It is also possible to create blocks by crashing gliders:

- Clear the screen.

- Load the brush "glidergun.life".

- Turn on the dragbar, if not already done.

- Place one copy of the glider gun at location 20:20.

- Turn right the brush two times with the rotation tool in the toolbar, i.e. the arrow pointing in a clockwise direction.

- Place the rotated copy at coordinates 104:100.

- Start the automata.

Two glider streams will be produced. The first gliders colliding in the middle of the screen will result in a block. The next pair of gliders will destroy this block.

Even more complex machineries are possible. Cause the simple "Life" application gets too slow for big configurations, let us switch to the less colorful "LightspeedLife".

- Open the application "LightspeedLife".

We need somewhat more room for the next demonstrations:

- Open the screen mode requester of the "Project" menu.

- Select the entry "Keep, but Med Res".

- Load the brush "RandomGun" and place it in the middle of the screen.

This is a rather big creation. It is a random number generator build with gliders, fishes, reflectors and two "switches".

Fishes are obstacles that travel like "gliders" with "light speed", but in vertical or horizontal directions. There are "fish factories" or "fish guns" like the glider gun we've seen before. Fishes come in three sizes, small medium and small. All of them are available separately as brushes, but you might want to discover them yourself.

The fish factory in this configuration is build in two pieces: The leftmost factory produces gliders that crash into another factory at the left top edge of the configuration to produce the fishes, traveling downwards.

Reflectors are "mirrors" for gliders. If one glider gets into a reflector at the right time, it changes its direction. Two reflectors are present here: One of them is at the right edge, and another one is a bit right and down from the second part of the fish factory, somewhat in the middle of the screen.

Two kinds of switches are used in the random gun. The first switch let fishes pass or anihilates them, depending of wether a certain block is present in this switch. It can be turned on or off by sending a glider in this block. It is located at the left edge, under the first part of the fish factory and is rather big. It consists of two almost similar parts sitting on top of each other. The "on-off" block is in the middle, near the right edge of the switch.

The second switch is the apparatus at the boundary of the random-gun. It is a somewhat different kind of the glider gun we discovered above, sending gliders in the right top direction. If a fish crashes into one of the gliders, the fish gets extinct and the glider changes its direction into the left top diagonal.

- Turn on the automata. Decrease the speed if things go too fast.

- Watch carefully. A random sequence of gliders will leave the random gun in the left-top diagonal direction.

How does it work?

The fish gun procudes fishes periodically. They pass the first switch below the fish factory or get destroyed, depending on the switch state. If a fish passes the switch, it turns the glider direction of the outcomming glider of the second switch from right to left, hence causing a glider to leave the random gun.

Gliders not leaving the automata gets reflected twice, into the first switch, causing a feedback.

Quite complex, huh?

The next experiment makes use of the fact that the "LightSpeedLife" application has no boundary, i.e. objects can move out of the plane. The experiment takes a while, depending on the computer power and presents an automata with increasing periods:

- Clear the screen

- Load the brush "HackSaw".

- Place it in the middle of the screen.

- Set the speed to maximum (i.e. the delay to zero).

- Start the automata.

You'll see an obstacle moving over the right boundary of the screen. Press the left arrow key to follow this object. It leaves a match type object behind it. This match gets light a few generations later some fishes crashing into it. The match itself burns faster than the obstacle creating it, hence reaching it somewhat later. At this point, the fire gets extinguished and the obstacle below the "match factory" sends a fish back to the leftmost machinery. Once the fish arived it, another flame gets produced that sets the match to fire again. However, the left obstacle had enough time to travel quite a big way to left, so this new cycle takes a lot of more time.

The last brush we're going to discuss is a "Life" implementation of "Hilbert's hotel". This mathematician wanted to illustrate the mathematical idea of Infinity and its quirks to people, by this little story:

Consider a hotel with infinetly many (but countably many) rooms. The hotel is "booked out", each room is occupied. The manager of a "finite" hotel has to reject new guests in this case, but that does not go for Hilbert's infinte hotel: If a new guest arrives, the guest of room one has to move to room two, the guest of room two to room three, guest three to room four and so on. Since there's no "last guest", everybody gets another free room after this re-organization, leaving room one free for the arrived guest.

This idea is now implented in LIFE, the guests are "Gliders" and the hotel is presented by four mirrors that move appart, making enough room for the next glider to arrive. The whole configuration is rather big, so it might happen that you don't have enough memory to run it. However, here is how it goes:

- Open the screen mode requester of the project menu.

- Select "Keep, but High Res". Do not press "O.K.". Enter "768" as both width and height of the screen.

- Load the brush "Hotel". Place it in the middle of the screen. Be careful! It has some tiny parts to the top left and right bottom of the main part which are easely overseen. Place it in the middle of the whole screen and make sure you haven't damaged these tiny bits. Make sure you press the button only once and you don't move the mouse while holding the button.

- Set the speed to maximum and start the automata.

The machineries at the left bottom and right top part in the middle of the hotel are traveling mirrors. They move appart and reflect the gliders acting as the guests of the hotel. All other parts generate guests in the right interval.

Quite a lot of other brushes are available for "LIFE". Try to experiment a bit! Most of the life brushes were converted from the Unix "XLife" brushes, try a look in the "XLife" directory of VideoEasel. The XLife source is plain ASCII, hence can be viewed with a standard editor (like "Ed"). Some of them come with comments explaining how they work. Have a look at these files, I can't discuss all of them since there are quite too many.

1.14 Going into the reverse gear

What's common to the automata "Brian's Brain" and "Greenberg" described above: Both use so called "second order" rules, the new configuration is constructed by the current configuration and the configuration before, which is kept by cells in the "refractory" state. Hence each new state is completely determinated by the previous two states. This characterizes a second-order system.

We're now describing a system in which the past enters the current state in a very simple way. Namely, the new state is the past XOR the present of the neighbours. It is a slight modification of the simple "Parity" automata from above.

But let's try some christmas stars first:

The experiment:

- Open the "Parity-Flip" application.

- Select the white color.

- Draw single dot in the center of the screen.

- Start the application.

The result is a fractal firework of starts. If you let this automata run for a while, you'll get your initial pixel back.

One interesting feature of this automata is that it is "linear", i.e. take two different configurations, start the automata separately for both configurations and "overlay" the result (XOR the colors). This will be simply the result of the overlayed start configurations.

Remember, this is very special and does not hold for, let's say, the LIFE application. Two pieces of a "glider" evolved give nothing, not two moved parts of a "glider".

Since the automata restored your single dot and a complex object consists of single dots, the automata must restore even complex start configurations by this "linearity". We'll check this with the next experiment:

- Clear the screen and select the white color.

- Draw some irregular shape with the polygone tool.

- Start the automata.

- You might recognize five copies of your initial shape after a while.

- After exactly 192 steps, the original configuration is restored.

We may even turn the automata from forewards into backwards:

- Clear the screen.

- Draw some irregular shape in white.

- Reset the generation counter.

- Start the automata and hold it after some steps.

- Write the generation counter down and clear it.

- Select "Swap Planes" from the picture menu.

- Restart the automata. It will restore your original configuration exactly after the same number of steps, by going thru the reverse states backwards.

Thus, we have access to the "reverse gear" of a time invariant system. We've also seen another property of time reversal systems: It is periodic, i.e. will restore the start configuration after a while. This does not only go for this system, but for all time reversal systems on a finite grid (this is basically a Theorem proven by the mathematician Poincare which will cause a lot of trouble once we're discussing lattice gases. But more will follow). However, the period isn't always this small!

How does it work?

The neighbourhood of a cell is this diamond shaped region:

*

+

*

A cell can be in one of four states: black, red, blue and white. The future of a cell depends only on its current state and on wether it has a even or odd number of cells either in the red or white state, everything else is ignored.

Everything else is more understadeable if we encode the colors in binary digits:

black : 0 0

red : 0 1

blue : 1 0

white : 1 1

^ ^

Hi Lo

The "Lo" part can be considered as the "current state" of the cell, the "Hi" part of the color number saves the past of the cell. The rule goes as follows:

Count the number of neighbours with a "current-on" state, i.e. red and white cells. If this number is odd, the cell in the center is turned "on", else "off". The last state of the cell isn't lost however, it is "shifted" into the "Hi"-part of the color number, hence a black cell with an odd number of neighbours gets "red", while a "red" cell with an even number of colors gets "blue". This implements the "memory" of the automata, by remembering the last state in the high-part.

1.15 TRON choreography

This is the first time we take the margolus neighbourhood into consideration. Since this neighbourhood definition is rather unique, please follow the link above if you read this first! The TRON automata itself is not very useful at all, but it presents an easy implementation of the margolus neighbourhood and creates patterns like in the "TRON" movie.

The experiment:

- Open the "VarTron" application. There's also a simple "TRON" which is almost like "VarTron", but more flickering and less colorful.

- Select the yellow color for drawing.

- Draw a rectangular frame with the rectangle tool.

- Start the automata.

The rectangle will start increasing or shrinking, forming some whirl like structures at some points. If it fills the complete screen, the automata starts to look more and more chaotic. However, it isn't in some sense "chaotic" at all, since we have a reverse gear! Here is how this magic "backwards switch" is engaged:

- Stop the automata.

- Choose the "Select Grid" menu item from the applications menu. It contains a checkbox saying "Odd grid", which is either checked or left blank. Click it in either case, reversing the grid choice of the margolus neighbourhood.

- Restart the automata. After a while, your initial frame will reappear, but maybe in a different color.

How does it work?

If you haven't read yet about the margolus rules, please do it now, as they are somewhat special. Here's the link.

Let us first forget about the color as it does not matter. You could have choosen the plain "TRON" as well, it shows basically the same behaiviour without this "extra".

The TRON rule itself is extremly simple: Invert each cell in a block except for the empty and the full block, which stay the same.

The reversibility is also easy to understand: This rule is obviously inverse to itself, i.e. applying it twice ON THE SAME GRID is as good as doing nothing. If you stop the automata and choose the other grid, the next step of the automata will be applied again ON THE SAME grid as the last step before you stopped the automata, inverting the last step. Then we find again the same grid of the step before the last. Applying the automata again undoes this generation again and so forth.

1.16 Critters swarms

This is another application of the margolus. This automata creates a rich structure of "bird like" objects that swarm across the screen. It might almost as interesting as LIFE, but hasn't been discussed so deeply. This might be a task for you...

The Experiment:

- Load the "VarCritters" application. There's also a plain "Critters" application which is black & white only and much more flickering, even though the flicker does not vanish completely with "VarCritters".

- Select the yellow pen for drawing.

- Draw some random yellow filled rectangle on the screen.

- Start the automata.

You'll see your rectangle deforming into an amorphus lump of matter. After a while, some tiny creatures start to excape from this blob and move in vertical and horizontal directions.

How does it work?

Let's discuss the plain "Critters" for simplicity. The "VarCritters" is almost like "Critters", however some efford was put in to remove the flicker of the original "Critters", using two identical pens - the two blue colors. If you change one of them to red, you'll see the same flicker of "Critters".

A cell in the "Critters" game has exactly two states, on and off. A block is complemented if it contains zero or four "on" cells, it is left unchanged if it contains two cells turned "on" and is complemented and rotated by 180° if it consists of three "on" cells.

Even though it doesn't look like this, this automata IS invertable, even though it doesn't look like this. However, unlike all automata we have seen before, the inverse automata is not the automata itself, so we can't go into the reverse gear that easy! Choosing the complement grid and the complement automata will turn back time. Finding the complementary automata is not so hard, so I leave this as an exercise to you!

1.17 Cellular computing

We turn back to "standard" automatas with non-standard use. We can actually model a complete computer system inside our cellular world! This idea goes back to von Neumann, who considered a cellular machine with 29 states with five neighbours each. Later on, this approach was simplified by Codd to a eight state cellular automata. This is still too complicate for us, so we use an even simpler solution developped by Banks.

A first experiment:

-Open the application "Bank's".

-Select the "Random Fill" option of the "Pics" menu.

-Start the automata.

The irregular noise starts to isolate to single islands connected by blue corners. From point to point some green or magenta dots appear.

Well, what did you expect: Throwing random micro chips in a box and soldering them doesn't give a computer! So some more systematic is needed...

We start with the simplest element of a computer: The wire!

The experiment:

-Clear the screen to remove the pile of junk.

-Load the brush "Wire".

-Place it somewhere on the screen.

-Start the automata.

Nothing happens! Remember, this is just a wire with no electricity in it. The serifes at the edges of the wire are required to protect the wire from erosion. Try to remove them, and restart the automata! But don't forget to restore the wire for the next experiment.

So let's stop the wire and add "one electron":

-Stop the automata.

-Choose the black color for drawing.

-Choose the "dots selector".

-Remove two pixel from the wire brush near the left edge until it looks like this:

*

*** **********...

** **********...

**********...

*

This represents one "right moving" electron on this wire.

-Start the automata.

The signal starts moving to the right and dies out when it reaches the right end of the wire. Signals can be made to move leftwards as well, of course. Simply draw the notch of the "electron" in the oppiside direction. If you rotate the wire by 90°, the signals will move upwards and downwards.

The next thing we build is a clock generator:

-Load the brush "Clock&Wire" and place it somewhere on the screen such that it does not intersect with the wire drawn before.

-Start the automata.

The clock on top of the wire will spit out one electron every eight cycles, and the electrons will travel downwards.

Clocks of different periods are available. Try "Clock_2&Wire" and "Clock_3&Wire". They look exactly like the eight period clock you've seen before, except that the "antenna" on top of the wire has a different size, which makes the period of the clock. However, it's not true that longer "antennas" make longer periods! The shorter "Clock_3&Wire" has a period of 16 which is the double period of the first clock.

Clocks itself aren't enough, it needs some gates to build a real computer. We present a simple "And Not" gate now: It provides two inputs, "A" and "B". If no signal is applied to input "B", the output of the gate is simply a copy of the input "A". If a signal is incomming at input "B" at the right time, the signal at input "A" is anihilated. Let's demonstrate this with the next brush:

-Clear the screen.

-Load the brush "AndNot_Gate" and place it somewhere at the screen.

-Start the automata.

The clock at the top edge of the screen generates one signal each sixteen cycles. The cloch at the right edge has the double period and ejects a signal each 32 cycles. Look at the gate in the middle of the screen: Each other signal of the clock on top of the screen passes the gate, while the remaining signals are eliminated by the "electrons" of the clock on the right.

Let's check what happens if we block the right clock:

-Select the blue color and click on the "dots selector".

-Draw one single dot on top of the outgoing wire from the right clock, like this:

*

****

****

****

-Restart the automata.

The signals from the right die on the blocker you created and don't reach the gate. Hence, the signals from the top clock will pass the gate without trouble.

This gate and a clock is actually all you need to build a computer. Some quirks remain:

We haven't seen yet how to make signals to turn corners and to cross each other. This can be figured out with a lot of patience, but shouldn't be done here. That's all needed to build a computer - in principle. Of course even the simplest circuit in a todays computer is way to big to fit on one screen, and the simulation would be way to slow. Anyways, this is only a pratical limitation, not a principle one.

How does it work?

Here's the rule of the game: Each cell has two states, on or off, black or colored. The color itself does not matter, but gives a fancier output. The rule how the next generation is generated is the following: Look at the von Neumann neighbourhood of the cell. If the cell lies in a "corner", erase it, if it is placed in a "pocket", fill it. To be precise, these are the rules:

1 0 1

011 -> 0 101 -> 1 101 -> 1

0 1 1

plus all rotated rules. If none of the three roles above apply, the center cell remains unchanged.