# Brief Tutorials of Trend

Trend

http://www.complex.iastate.edu

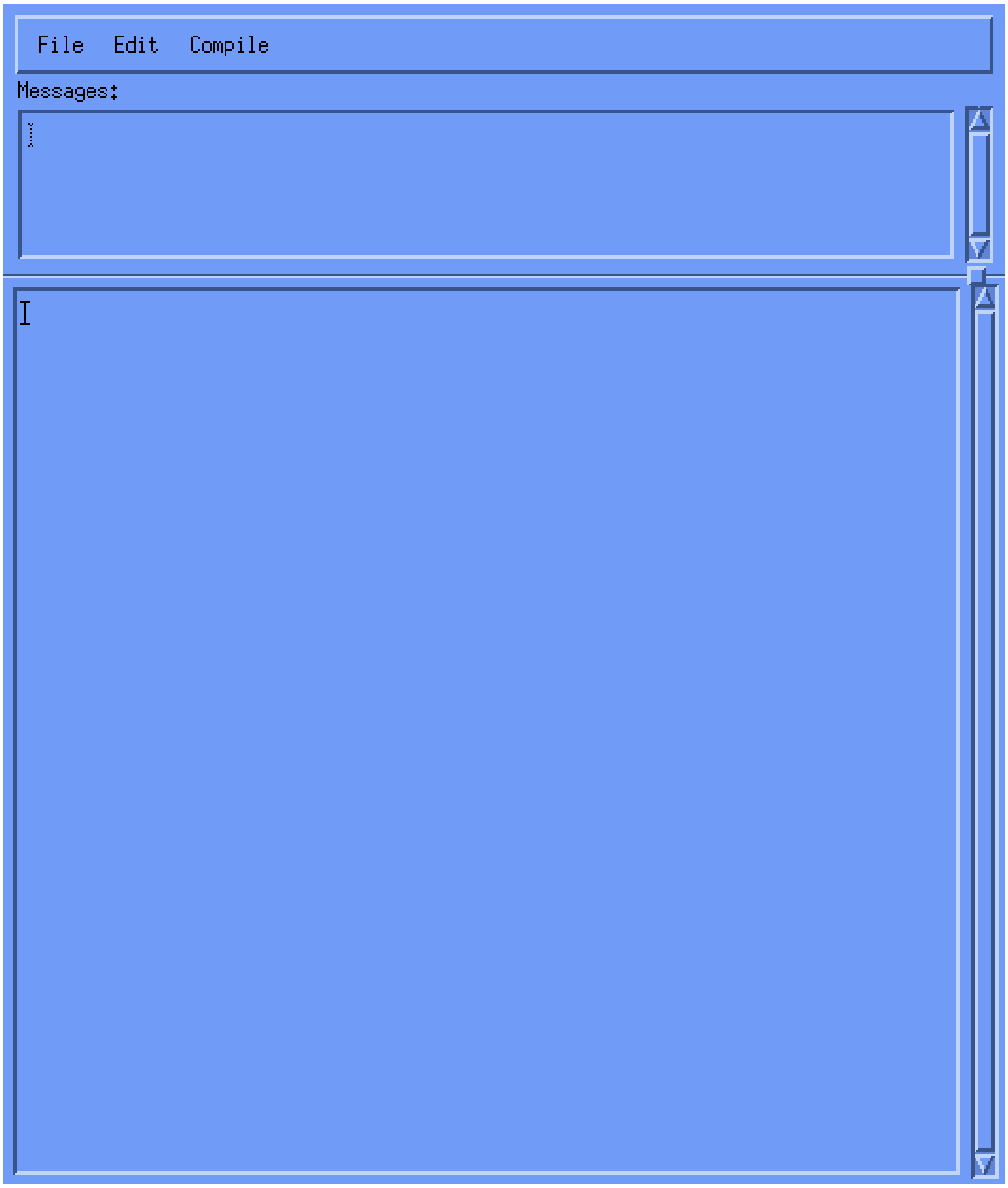
**Iowa State University Complex Computation Lab**
**July, 2001**

# Basic TREND Tutorial

## 1. Windows in TREND

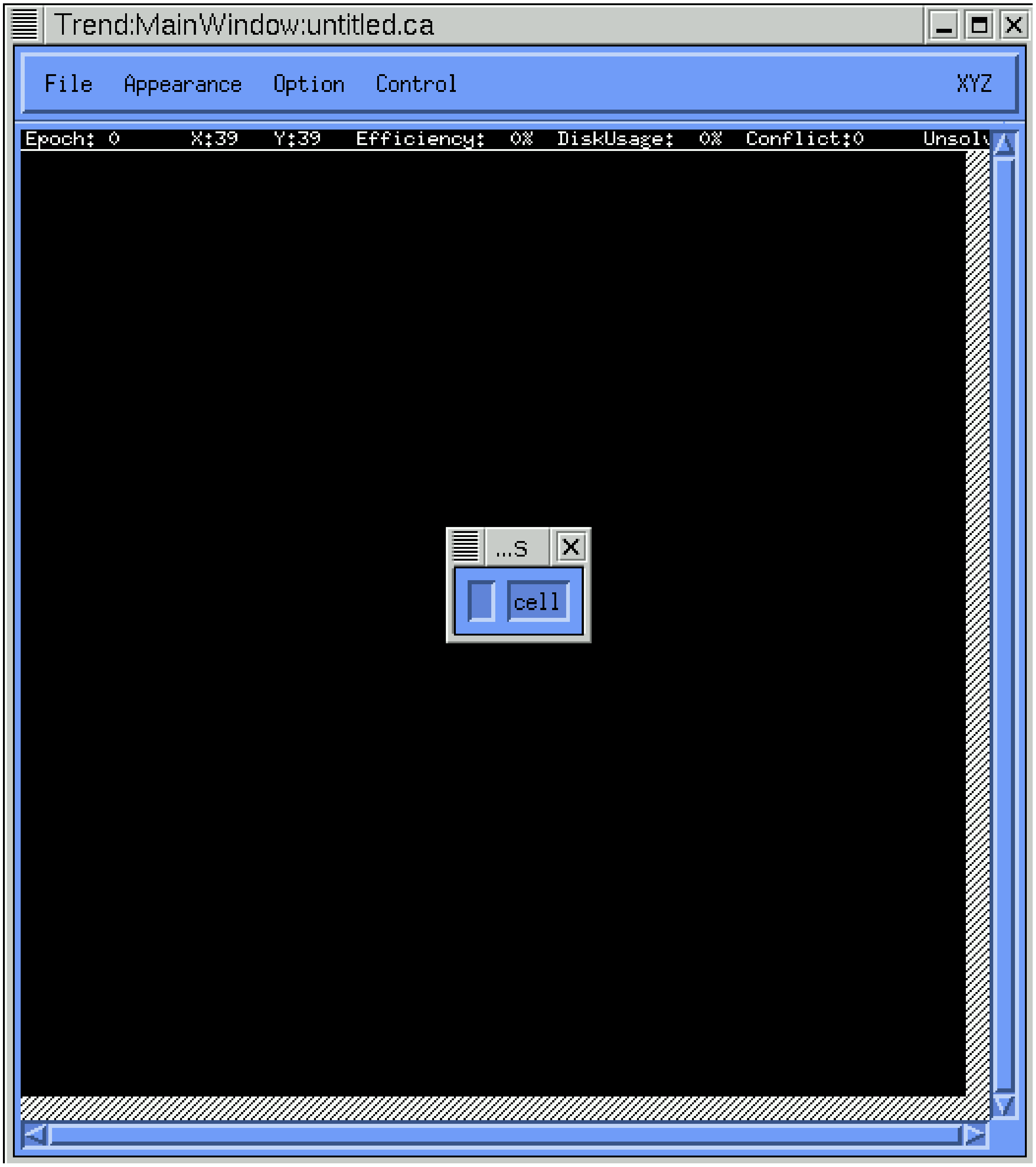
Before you begin, it is important to understand the tools you will use while working with TREND. After starting the TREND application, three main screens will appear layered on top of each other. The initial screen that appears on top is the .RULE screen.

***Below is the .RULE screen.***



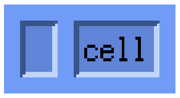
Click and drag the .RULE screen to a side so the .CA screen is visible.

***Below is the .CA screen.***



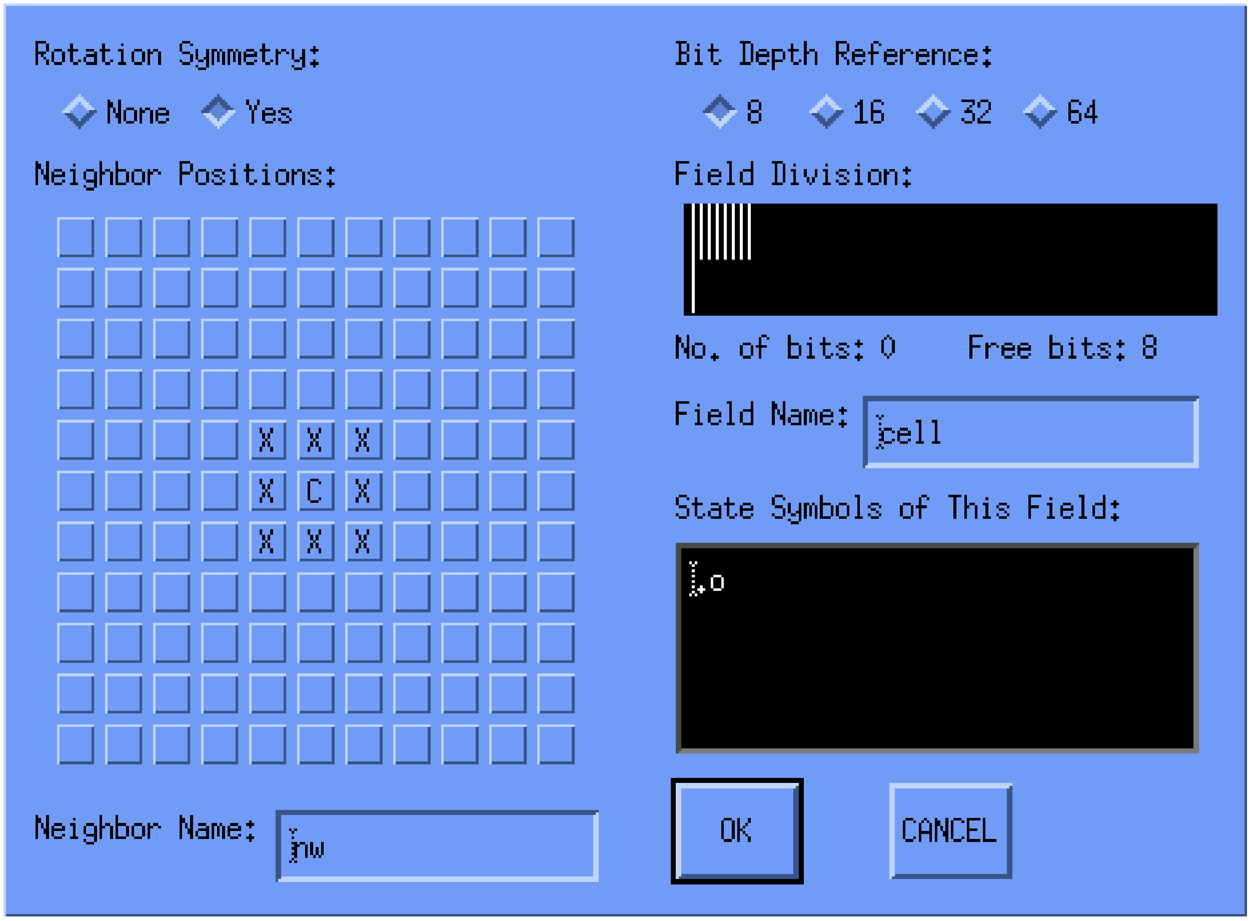
```
Epoch: 0      X:39   Y:39   Efficiency:  0%  DiskUsage:  0%  Conflict:0     Unsolv
```

Within the .CA screen is another smaller screen that shows the current fields. The default is one field called *CELL*. This will be discussed more later. For now click and drag the small screen to a free area on the desktop so it is not on top of the .CA screen.



Arranging the screens is similar to most applications. The option of resizing the screens is available by moving the mouse to the border of the individual screens and dragging the borders until the size desired. For now arrange the three separate screens so they all are visible on the desktop.

When starting a new project, there is the option of changing the default for the fields. As noted earlier, the default for the initial field called *CELL* is one bit that can hold 2 state symbols. Select the **File** option on the CA screen and select **Modify template...** The screen will appear as it does below.



No. of bits: 0 Free bits: 8

Field Name: cell

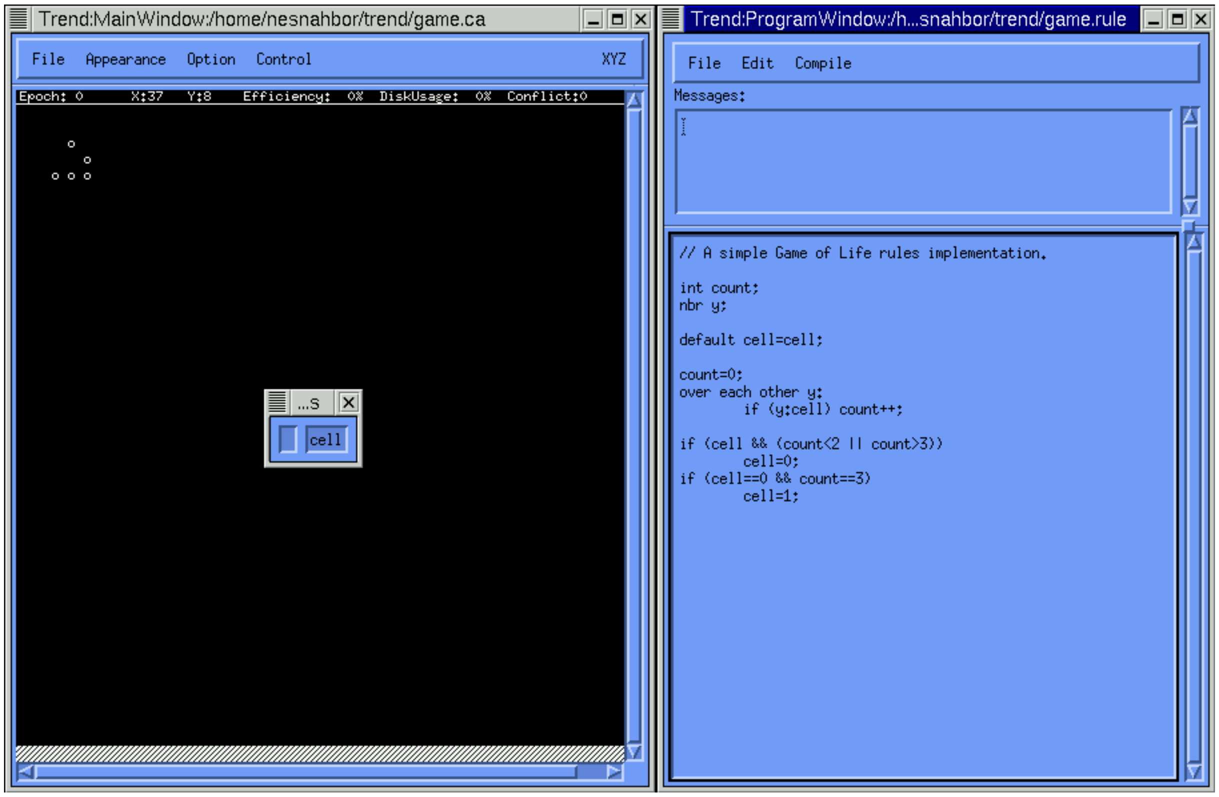
State Symbols of This Field: o

Neighbor Name: nw

For now just click **Cancel** and return to the initial CA screen

## 2. Controls in TREND

works with cells. Each cell is separate and the code has the ability to affect each cell separately. That is the idea of **Cellular Automata**. For that reason, **TREND** uses a CA space. It is a screen that is broken up into cells, or a kind of grid. Each location on the grid is a cell. Symbols can be used to create a unique CA space, or to recreate an existing CA space. In the example below, the shape for **The Game of Life** is loaded into the CA space. This shape can be modified by using the right-mouse click to select or unselect a particular symbol for a specific cell location on the CA space.

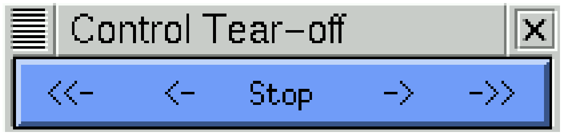

Trend:MainWindow:/home/nesnahbor/trend/game.ca
File Appearance Option Control XYZ
...S
cell
Trend:ProgramWindow:/h...snahbor/trend/game.rule
File Edit Compile
Messages:

```
Epoch: 0    X:37   Y:8    Efficiency:  0%  DiskUsage:  0%  Conflict:0
```

```
// A simple Game of Life rules implementation.

int count;
nbr y;

default cell=cell;

count=0;
over each other y:
        if (y:cell) count++;

if (cell && (count<2 || count>3))
        cell=0;
if (cell==0 && count==3)
        cell=1;
```

**TREND** allows the user to control the compiled rule step by step. With a set of controls, the user can control the *epoch* of the CA space. The *epoch* is like the number of cycles, or the number of times the rule has been applied to the shape in the CA space. The *epoch* number is shown in the upper-left hand corner of the CA space, just below the menu bar. *Epoch* 0 is the starting point.

To advance the *epoch* the user has the option of the on-screen controls or the keyboard controls. To use the on-screen controls, select **Control** from the CA screen menu. The box below will appear.



To have this option on the screen at all times, select the **Controls** option again and then click the dotted line. This will allow you to "tear off" the Controls box and move it to a open portion of the desktop for later use. The function of the buttons are shown in the chart below. If the user prefers to use the keyboard to control the rule, the corresponding arrow buttons are shown as well.

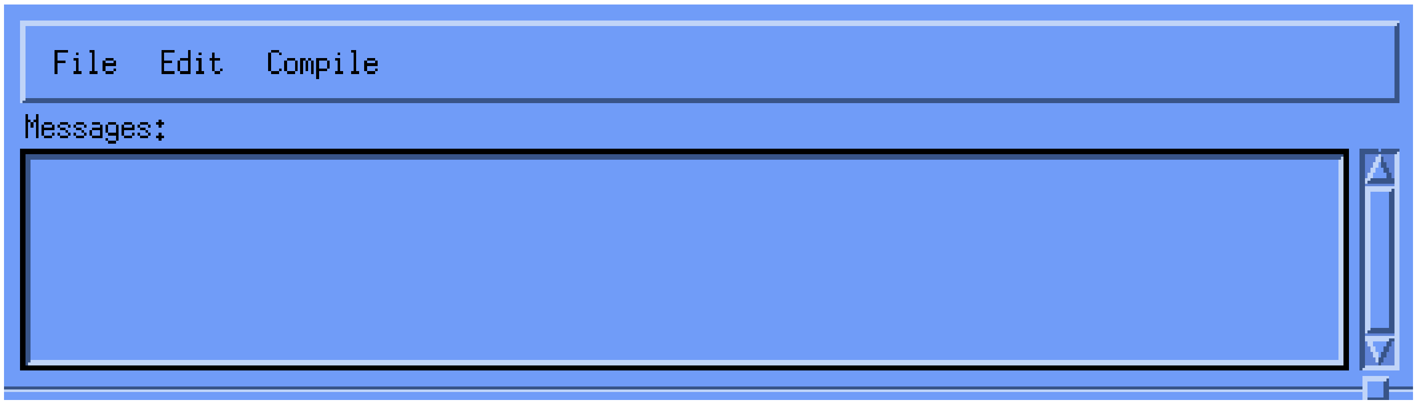
To have this option on the screen at all times, select the **Controls** option again and then click the dotted line. This will allow you to "tear off" the Controls box and move it to a open portion of the desktop for later use. The function of the buttons are shown in the chart below. If the user prefers to use the keyboard to control the rule, the corresponding arrow buttons are shown as well.

| **button** | **action** | **keyboard** |
|---|---|---|
| -> | forward | right-arrow |
| ->> | fast forward | up-arrow |
| stop | stop | spacebar |
| <- | back | left-arrow |
| <<- | rewind | down-arrow |

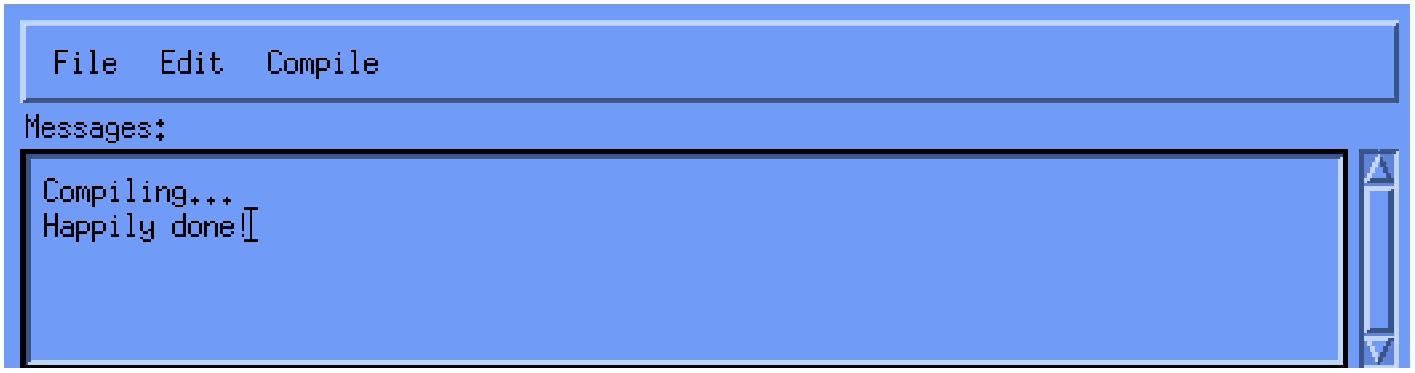
One last important note on use of the controls. It is a "good idea" to return the *epoch* number back to zero before making any changes in the rule. This will allow the original shape to be tested by the changes in the rule after compiling. So a "rule of thumb" to follow is to rewind the CA space each time before making any changes.

## 3. Rule Compiling in TREND

When using TREND, the code must be compiled before the rule can be run. This is the case on a rule just written, or an existing rule that has been loaded from memory. Because you can use a particular rule with an infinite number of CA space configurations, the rule must be compiled with relation to a particular CA space and .tmpl that has been chosen/created for this rule to be executed with.

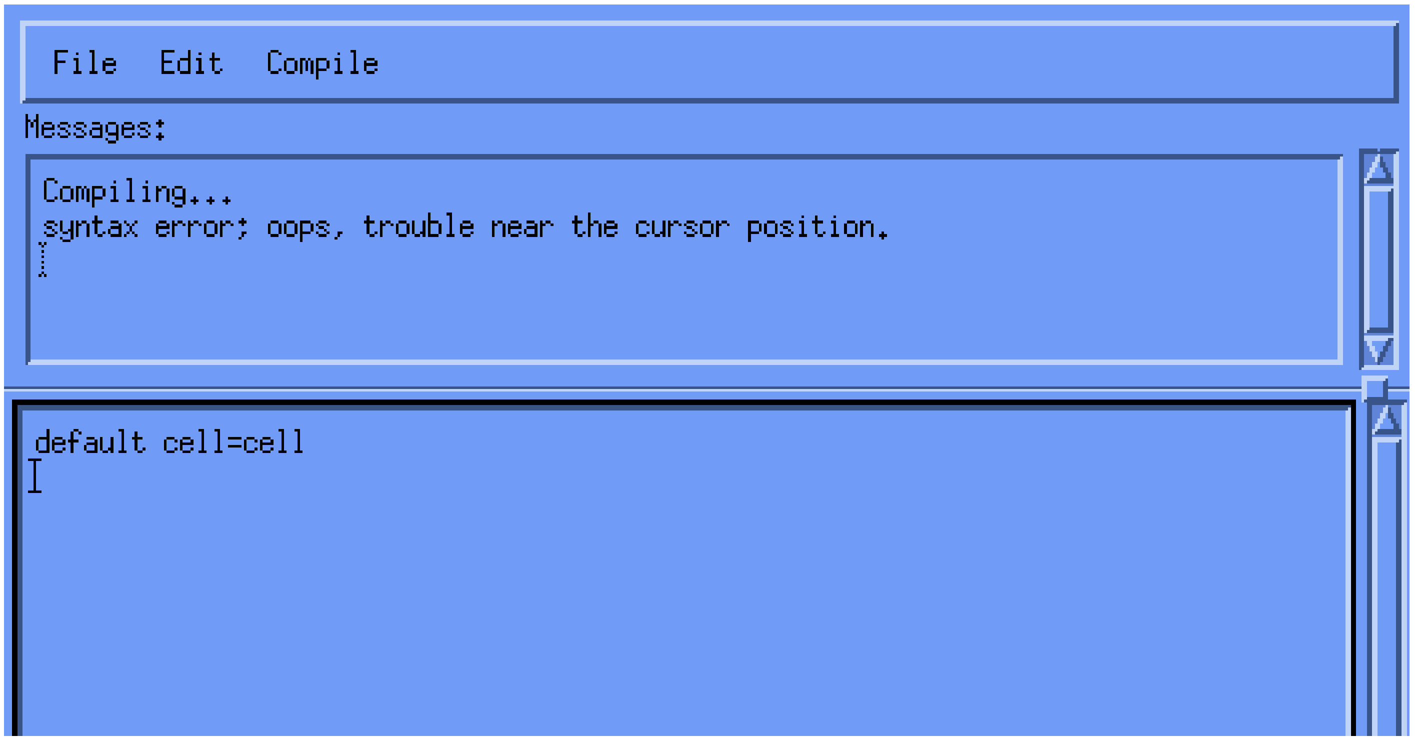


At the top of the coding screen is an option labeled *Compile*. Select this option and select *Now*. The compiler will attempt to compile the code in the *.rule* screen below. If it is successful, it will appear as the example does below.



```
Compiling...
Happily done!
```

As with most languages, any changes that have been saved in the coding screen must be compiled again in order to be recognized. All rules provided by the downloaded version of TREND will compile with no errors if used as they are provided.
In the example below, the code has a simple error.



```
Compiling...
syntax error: oops, trouble near the cursor position.
```

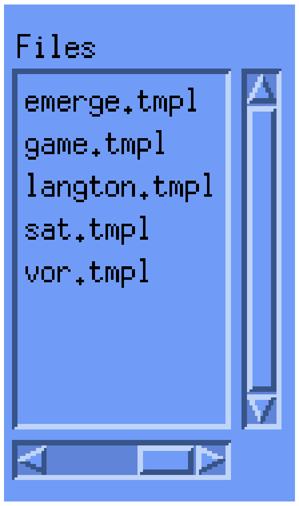
```
default cell=cell
```

In the only line of the code, the semicolon is missing to signify the end of the line. When compiled, an error message is provided and the cursor will move to the general location of the error. In this case, there is only one line, but the cursor moves to the end of that line to signify where the problem may be. Forgetting the semicolon is a common error in TREND as it is in C++.

# The Game of Life

## 1. Loading an existing rule

The first example project to examine is called The Game of Life. To load this project, select the File option from the .CA screen and select Get template... In the downloaded version of TREND, there are existing rules that are made available to the user. In the section called Files (shown below) select the game.tmpl. This will be the first example of an existing rule to examine. Disecting this rule willhelp in understanding how to code in TREND.

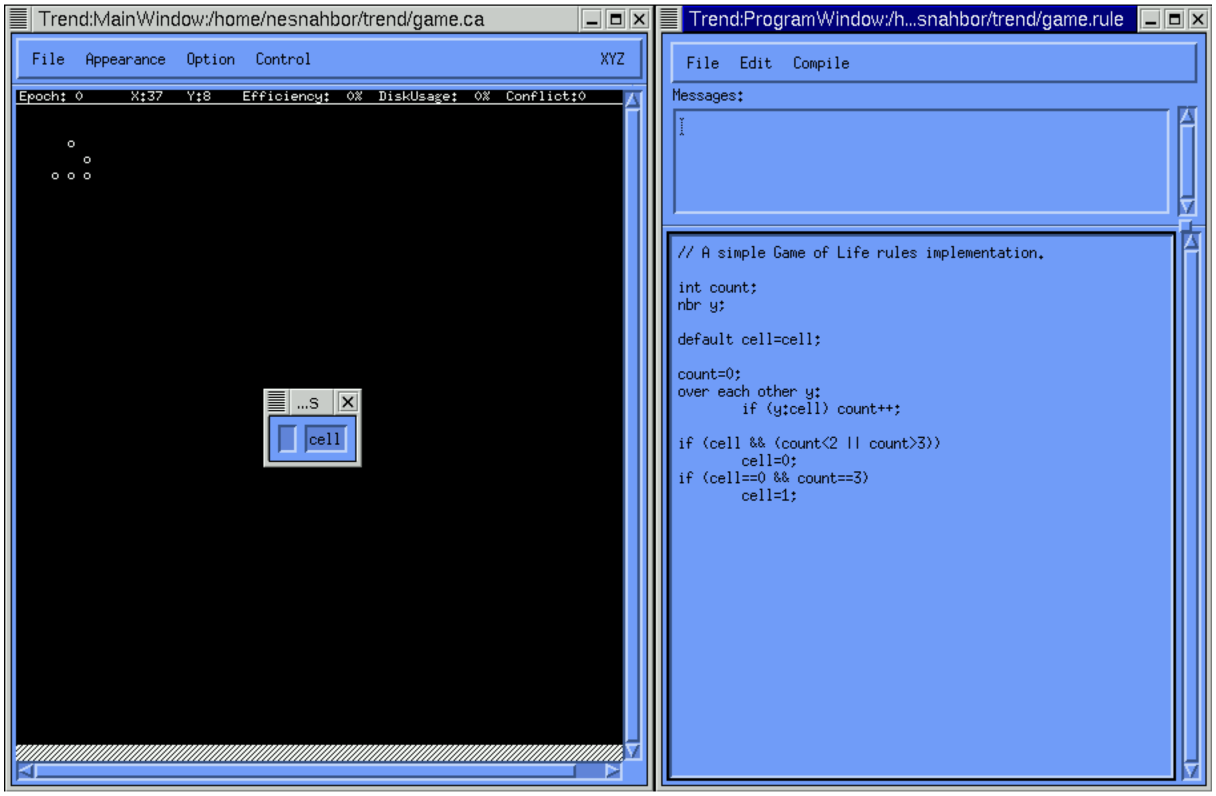

Files

emerge.tmpl
game.tmpl
langton.tmpl
sat.tmpl
vor.tmpl

Once the game.tmpl has been opened with the Get template... option in the .CA screen, TREND will open the game.rule and game.ca screens as well. This is the case only as long as they all exist in the same directory. This is seen in the path labeled at the top of the screens below. You will see that for this example, the path is /home/nesnahbor/trend/game.ca. In this case, the game.tmpl, game.rule and game.ca all exist in the same directory. For each user the path will vary depending on the user name and where the files are stored. Click and drag the screens to the empty portions of the desktop so all three screens are visible as Figure 1.

## 2. Examining the rule

The code (rule) for **The Game of Life** is now in the *.rule* screen. The syntax of the code is similar to that of a C++. The lines with nothing in them are ignored, as are the lines with comments. The indentations are up to the user, but it is recommended to use indentation to keep track of structure. Below is a numbered version of the same rule shown above to make it easier to examine each line of the code.


Trend:MainWindow:/home/nesnahbor/trend/game.ca
File Appearance Option Control XYZ
...S
cell
Trend:ProgramWindow:/h...snahbor/trend/game.rule
File Edit Compile
Messages:

```
Epoch: 0     X:37   Y:8    Efficiency:  0%  DiskUsage:  0%  Conflict:0
```

```
// A simple Game of Life rules implementation.

int count;
nbr y;

default cell=cell;

count=0;
over each other y:
        if (y:cell) count++;

if (cell && (count<2 || count>3))
        cell=0;
if (cell==0 && count==3)
        cell=1;
```

Figure 1.   The windows of Trend

| **line#** | **code** |
|---|---|
| 1 | // A simple Game of Life rules implementation. |
| 2 | |
| 3 | int count; |
| 4 | nbr y; |
| 5 | |
| 6 | default cell=cell; |
| 7 | |
| 8 | count=0; |
| 9 | over each other y: |
| 10 |   if(y:cell) count++; |
| 11 | |
| 12 | if (cell && (count<2 \|\| count>3)) |
| 13 |   cell=0; |
| 14 | if (cell ==0 && count==3) |
| 15 |   cell=1; |

**Trend** has the ability to add comments in the same way as C++. In line #1, the line starts with **//** which allows comments to be added without affecting the code. If the comment is long enough to continue to the next line, the **/*...*/** is an option as well. Comments are suggested in order for others to follow the rule structure and for later debugging if necessary. The lines with no code are just for cosmetic purposes to make the code more readable.

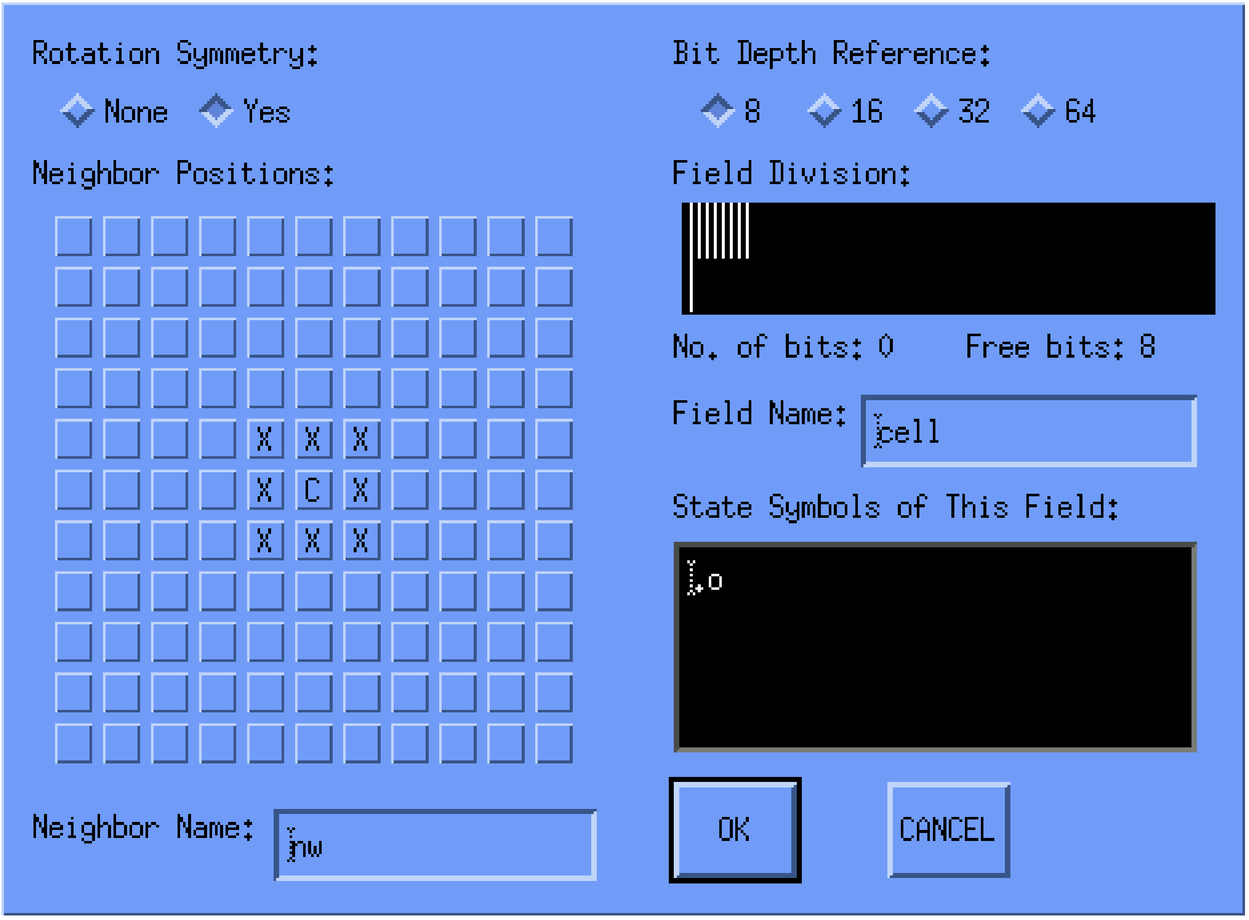
```
1 // A simple Game of Life rules implementation.
```

In this case, the comments are used as a short description of the rule.

```
3 int count;
```

Line 3 is a declaration of an integer named **count**. Again this is similar to the C++ programming language and declaring variables. In this rule, the integer **count** is used as a counter for the number of neighbors that are currently "alive". This will be explained more as the theory behind **The Game of Life** is explained later in the documentation.

```
4 nbr y;
```


Rotation Symmetry: None Yes

Bit Depth Reference: 8 16 32 64

Neighbor Positions:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | X | X | X | | | | |
| | | | | X | C | X | | | | |
| | | | | X | X | X | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |

Field Division:

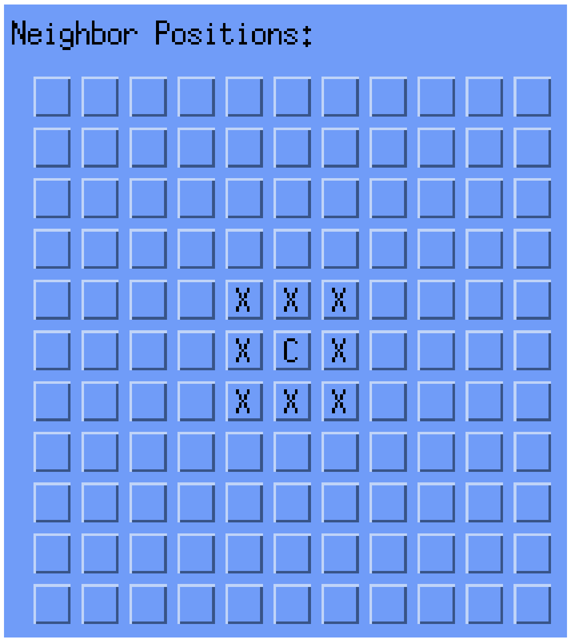
No. of bits: 0 Free bits: 8

Field Name: cell

State Symbols of This Field: .o

Neighbor Name: nw

Looking at the neighbor positions



Line 4 is a declaration of a nbr type named **y**. The nbr type refers to a cell's neighbors which are defined in the *.tmpl screen* shown below.The rule **The Game of Life** has 8 neighbors around each cell. This is shown above with the **C** being a particular cell and **X** being the neighbors. The neighbors can be modified to include up to 32 neighbors. This will be used with other rules explained later. The code in line 4 will check all the neighbors to see if they are "alive", have a value of something other than zero. If a neighbor is "alive", it will increment the **count** variable. An example would be if a particular cell **"C"** has 5 neighbors **"X"** that are "alive", it will increment the **count** variable to 5. The value of the **count** variable determines the effect of the rule on the cell **"C"**. This will be more obvious once the rest of the lines of code are explained.

```
6 default cell=cell;
```

Line 6 is a sort of standard declaration in **TREND**. It suggests that if no rule applies to a particular cell, then leave it as is. For example, if you have a CA space that has only one "alive" cell, and your rule requires more than one "alive" cell in order to execute the statement, nothing will happen and the CA space will not change.

```
8 count=0;
```

Line 8 assigns a value to the variable **count**. Again this is similar to programming in C++.

```
9 over each other y:
```

Line 9 is the command that allows each neighbor of a particular cell to be examined for the state of that neighbor cell. It allows for different neighbors to be specified. For example, a cell may have a ***no*** (North) neighbor that physically exists above the original cell. Or a neighbor that exists underneath the original cell may be called the ***so*** (South) neighbor. So each neighbor has a specific name in order to keep them apart and work with them separately. The actual names can be user defined by using the *.tmpl* screen talked about earlier and discussed further in the tools section.

```
10 if(y:cell) count++;
```

Line 10 is a traditional "if" statement with the condition and a statement to follow when the condition is true. In this case, the code reads **if(y:cell)**. Which means any neighbors of the original cell (think back to the **C** surrounded by the **X** as neighbors) that are "alive", then increment the variable **count** by one. Going back to an earlier example, if 5 neighbors are "alive", then increment the variable **count** 5 times. Since it started with a value of 0 by line 8 in the code, incrementing it 5 times will give **count** a value of 5.

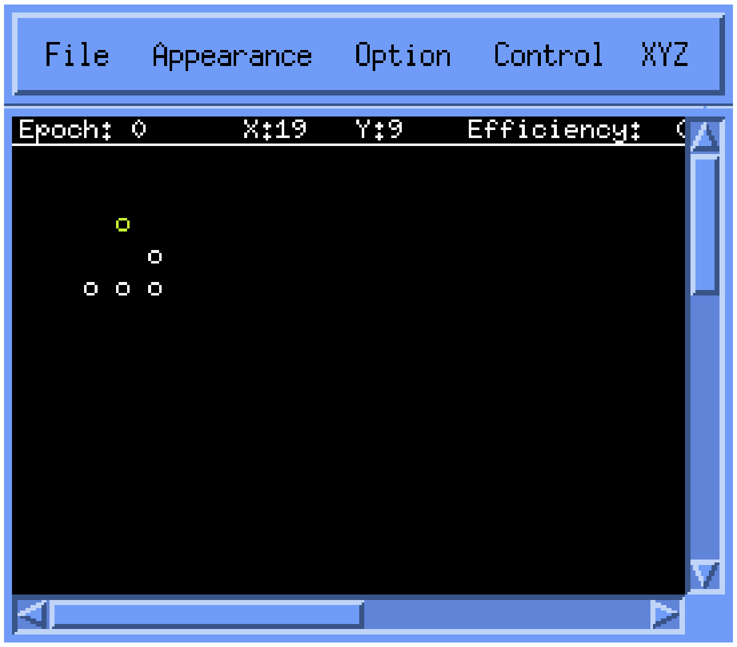
```
12 if (cell && (count<2 || count>3))
13   cell=0;
```

Line 12 is a conditional statement that starts out by saying **if cell**. That is an easy way of stating a condition that says if a particular cell has a value other than 0, then it is "alive" and can be called **cell**. If a **cell** has a value of 0, then it is considered "dead" and is shown as **cell==0**. So back to line 12. It states that if the cell is "alive" and the number of it's neighbors that are "alive" is <u>less than 2</u> or <u>greater than 3</u>, then execute the statement that follows. That statement is shown in line 13. It says assign a value of 0 to the cell, or in other words make it "dead". So to recap lines 12 and 13. The condition states that if any cell has a value other than 0, <u>and</u> it has less than 2 "alive" neighbors <u>or</u> more than 3 "alive" neighbors, assign the value of the cell to 0, making it "dead". This rule is simulating the theory of **The Game of Life** in a sense. The theory is if a person exists and has too few neighbors to interact with, that person will die. Likewise, if a person exists and they have too many neighbors, they will get crowded out and die.

```
14 if (cell==0 && count==3)
15   cell=1;
```

Line 14 is the last condition for this rule. It states that if a cell is "dead" (has a value of 0), and has 2 or 3 neighbors that are "alive" (values other than 0), assign the "dead" cell a value of 1. Making it "alive". Going back to the theory of **The Game of Life**, the rule simulates the idea that if you have 3 neighbors, you will have some sort of reproduction or division that takes place to produce more cells. This rule can be changed slightly to get dramatically different results.

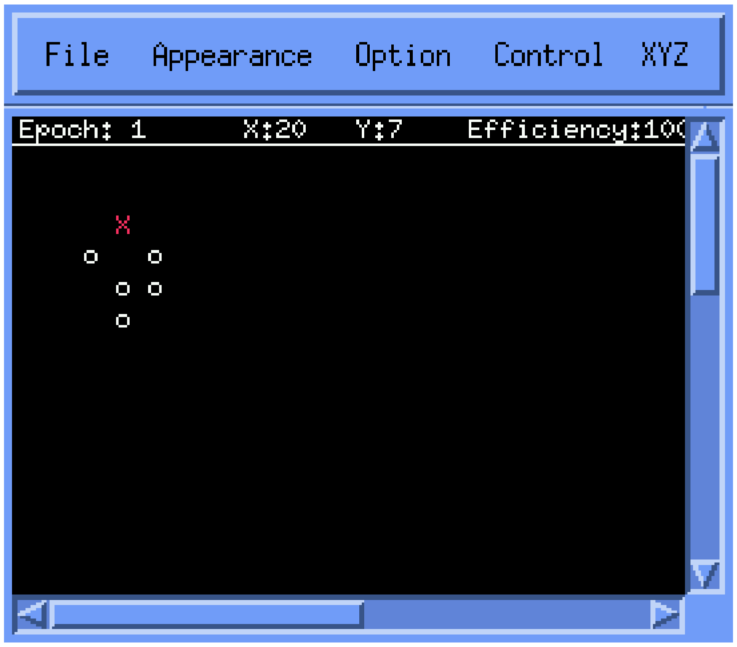
## 3. The RULE in action



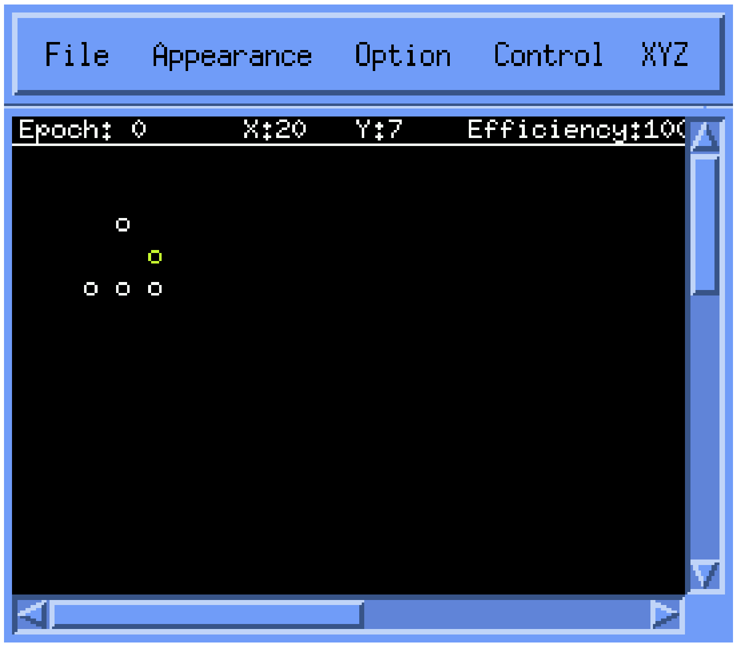
Looking at the rule again one cell at a time to see how the rule affects each symbol in the CA space. First symbol highlighted is the top cell with the green color. This cell is highlighted to show the affects of the rule once it is applied. The cell is "alive" and has only one neighbor that is "alive". That neighbor is the cell to the lower right of the green cell. So according to the rule, lines 12 and 13, this cell colored in green will be assigned a value of 0.

```
12 if (cell && (count<2 || count>3))
13   cell=0;
```

After clicking the controls to advance the rule one time, the green cell is assigned a value of 0 and is considered a "dead" cell at this point. The green cell has been assigned a value of 0, therefore it no longer shows on the CA space. A red X has been placed in the exact spot in order to show the movement of the symbols after one epoch of the rule has been run.



Return the CA space back to it's original configuration with the *rewind* button (<<-)or *down-arrow key* as discussed earlier in the controls section.

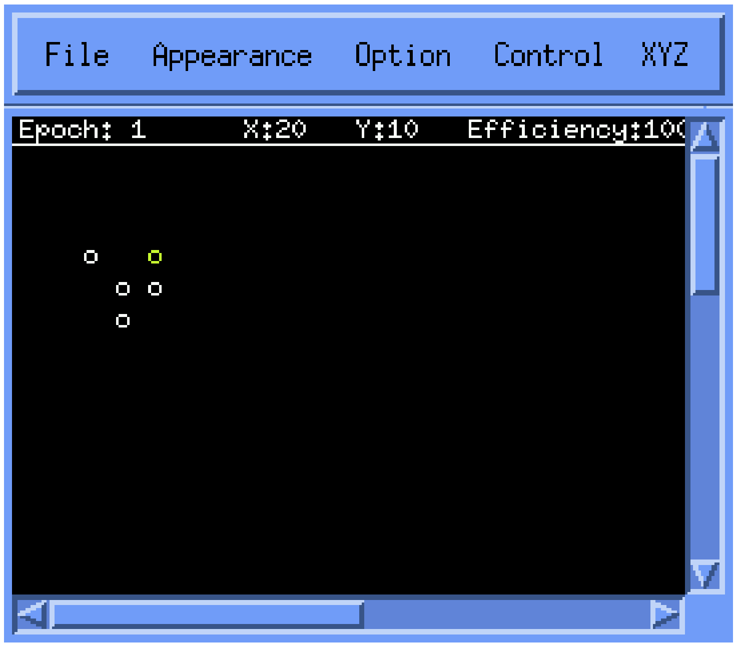


This time a different cell has been highlighted with green. This particular cell is "alive" and has 3 neighbors that are "alive". Therefore the section of the rule that applies to this cell are lines 12 and 13. The green cell has a value of something other than 0, and this time it has a count value that falls between count<2 and count>3. So the condition is proven 'false' and the cell is left alone for atleast this epoch. This is the case because of line 6 that says if no other rule applies to this cell, leave it as is.

| | |
|---|---|
| 6 | default cell=cell; |

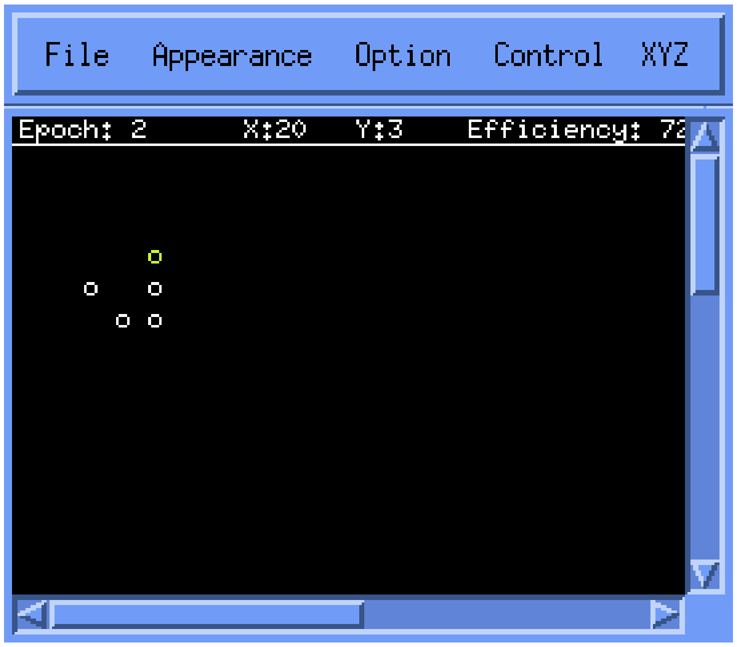
Advancing to epoch 1 shows the cell remains "alive" once the rule was applied to

the CA space for the first epoch. It is highlighted in green on the image below. Notice other cells have been affected by the rule, but for now the green cell has been unaffected after one
epoch.



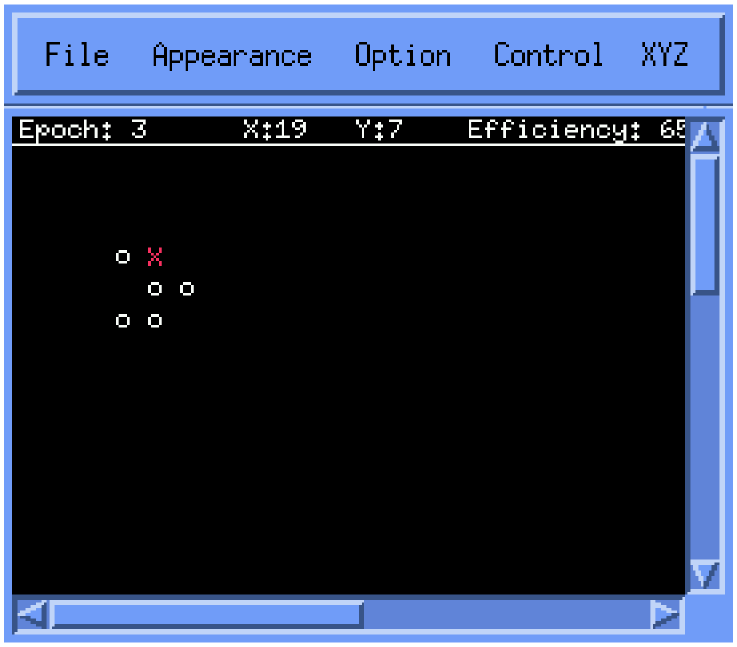
Epoch: 1 X:20 Y:10 Efficiency:10

However, the situation is now a bit different for the same cell. It is still an "alive" cell, but it no longer has exactly 3 "alive" neighbors. After epoch 1, the cell only has 2 "alive" neighbors. Again the number of neighbors falls within the condition stated in line 12, so this cell is again left alone. Below is the result after the second epoch.



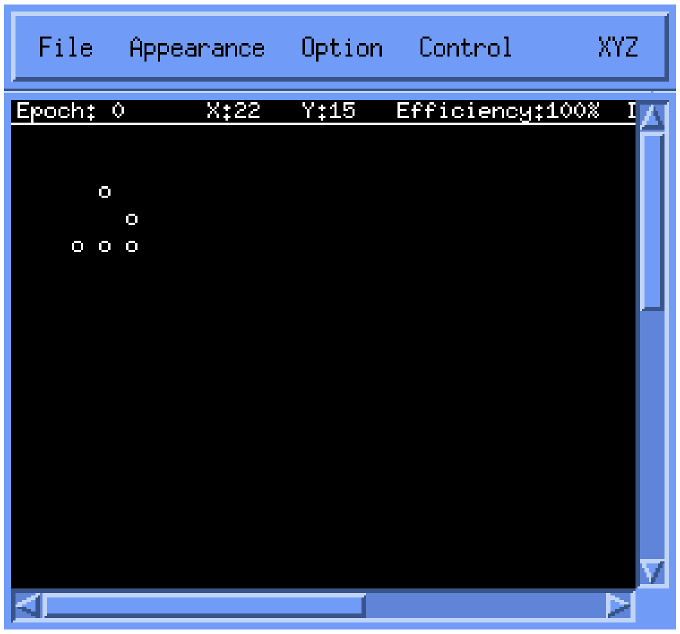
Epoch: 2 X:20 Y:3 Efficiency: 72

Continuing on to the next epoch, and concentrating on the same green highlighted cell, we can see the next epoch will "kill off" the cell because the number of "alive" neighbors is below 2. So the condition in line 12 of the code is proven true and the statement in line 13 is executed. So the cell is assigned the value of 0 and "killed off". The red X is added as a reminder of the cell location in the CA space. The red X has nothing to do with the rule, but has just been added for visual support for positioning.



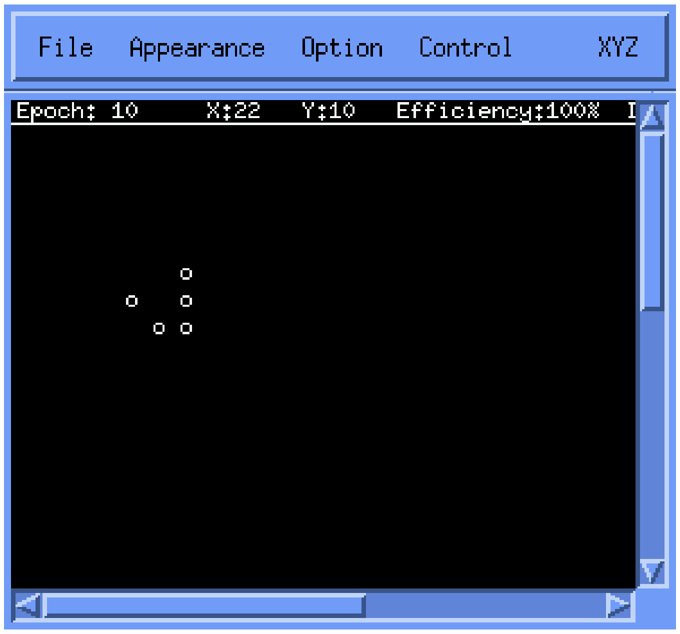
Epoch: 3 X:19 Y:7 Efficiency: 65

After each epoch, the rule would continue to be applied to the cells in the CA space. Occasionally the combination of the rule and the configuration of the symbols in the CA space result in a pattern or trend. In the case of **The Game of Life**, the pattern for this particular set of symbols in the CA space results in what appears to be movement to the lower right-hand corner. The following images show the changes at certain intervals. Be sure to look at the epoch number of each image to have an idea of how many epochs must occur to get the resulting shape.



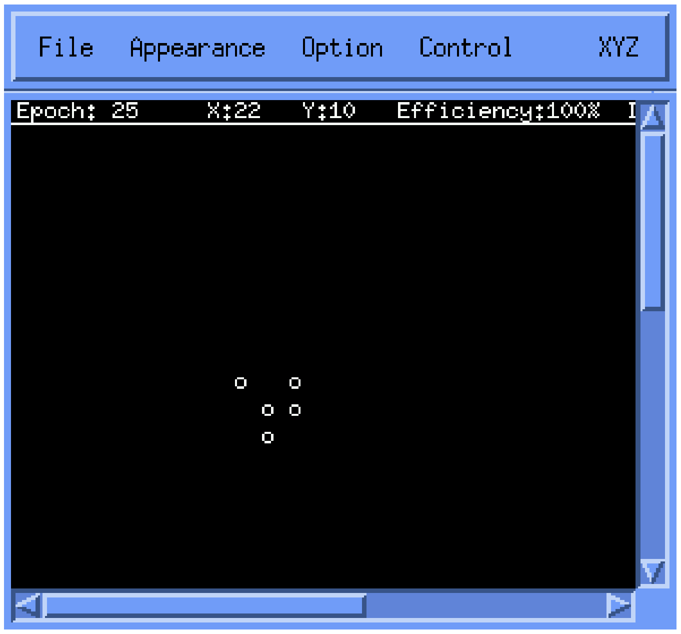
Epoch: 0 X:22 Y:15 Efficiency:100% I

***Epoch 0***



Epoch: 10 X:22 Y:10 Efficiency:100% I

***Epoch 10***



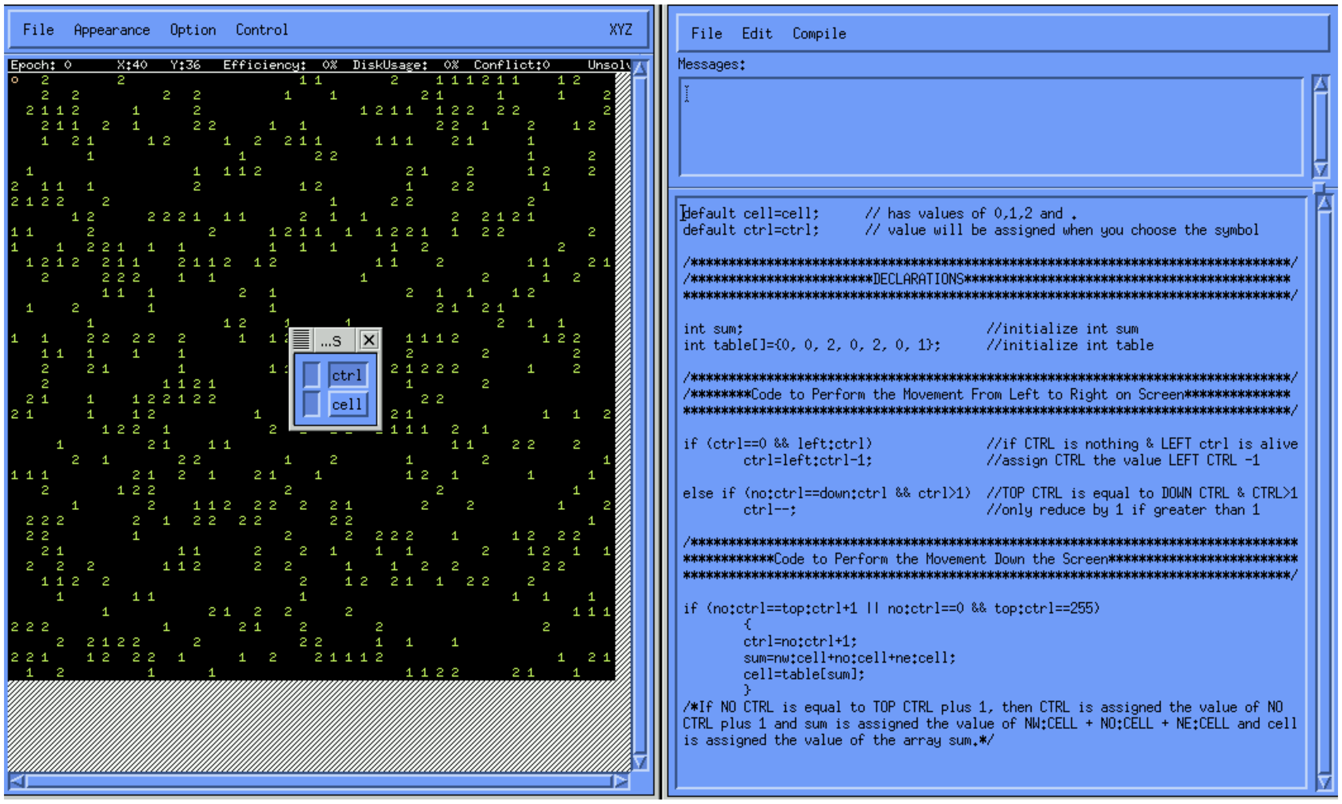
***Epoch 25***

Try selecting the ->> or the *up-arrow key* and let the rule run. The symbols continue to move across the screen until it is stopped with the controls. This has commonly been referred to as a "crawler". There are many rules that when combined with certain set of symbols, can create the impression of movement across the CA screen.

# One Dimensional Cellular Automata

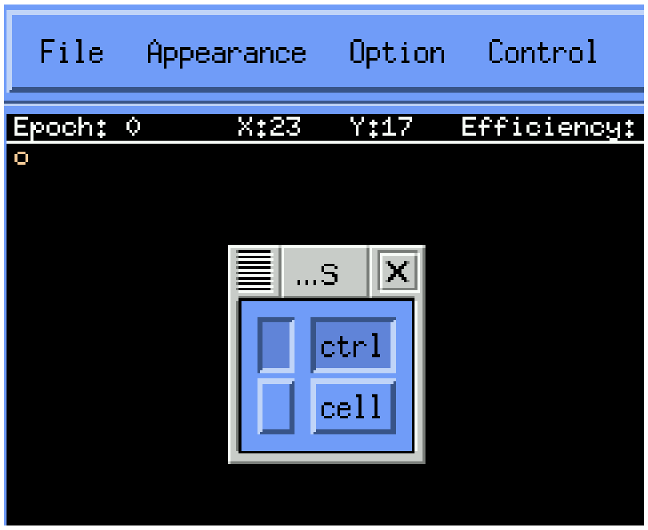
## 1. Loading an existing rule

The second example project to examine is the **One Dimensional Cellular Automata**. This project is also available with the download of TREND. Once the project has been downloaded, select the **File** option from the .CA screen and select **Get template...** Navigate to the local directory in which the project has been saved. Next, find and select the project named **model.tmpl**. TREND will open the *.rule* and *.ca* screens as well. This is the case only as long as they all exist in the same directory. Click and drag the screens to the empty portions of the desktop so all three screens are visible as they are below.

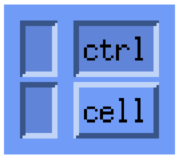


```
default cell=cell;      // has values of 0,1,2 and .
default ctrl=ctrl;      // value will be assigned when you choose the symbol

/*******************************************************************************/
/************************DECLARATIONS*****************************************
*******************************************************************************/

int sum;                              //initialize int sum
int table[]={0, 0, 2, 0, 2, 0, 1};    //initialize int table

/*******************************************************************************/
/********Code to Perform the Movement From Left to Right on Screen*************
*******************************************************************************/

if (ctrl==0 && left:ctrl)             //if CTRL is nothing & LEFT ctrl is alive
        ctrl=left:ctrl-1;             //assign CTRL the value LEFT CTRL -1

else if (no:ctrl==down:ctrl && ctrl>1)  //TOP CTRL is equal to DOWN CTRL & CTRL>1
        ctrl--;                       //only reduce by 1 if greater than 1

/*******************************************************************************
***********Code to Perform the Movement Down the Screen************************
*******************************************************************************/

if (no:ctrl==top:ctrl+1 || no:ctrl==0 && top:ctrl==255)
        {
        ctrl=no:ctrl+1;
        sum=nw:cell+no:cell+ne:cell;
        cell=table[sum];
        }
/*If NO CTRL is equal to TOP CTRL plus 1, then CTRL is assigned the value of NO
CTRL plus 1 and sum is assigned the value of NW:CELL + NO:CELL + NE:CELL and cell
is assigned the value of the array sum.*/
```

In this example, the coding window has been enlarged a bit in order for the whole rule to be shown. It is recommended to include comments within the code. As was stated in the first example for **The Game of Life**, comments can be initiated with a *double slash* **(//)** as well as *slash asterick* **(/*)...(*/)** for comments extending more than one line.



Note that the shape on the CA screen is different in this example than it was in **The Game of Life** example. Here there is only one symbol (character) showing in the upper-left hand corner of the CA screen. This is misleading because only the *ctrl* field is selected. The *cell* option in this example has not been selected. Select the *cell* option from the field box so that both boxes are selected as shown below.

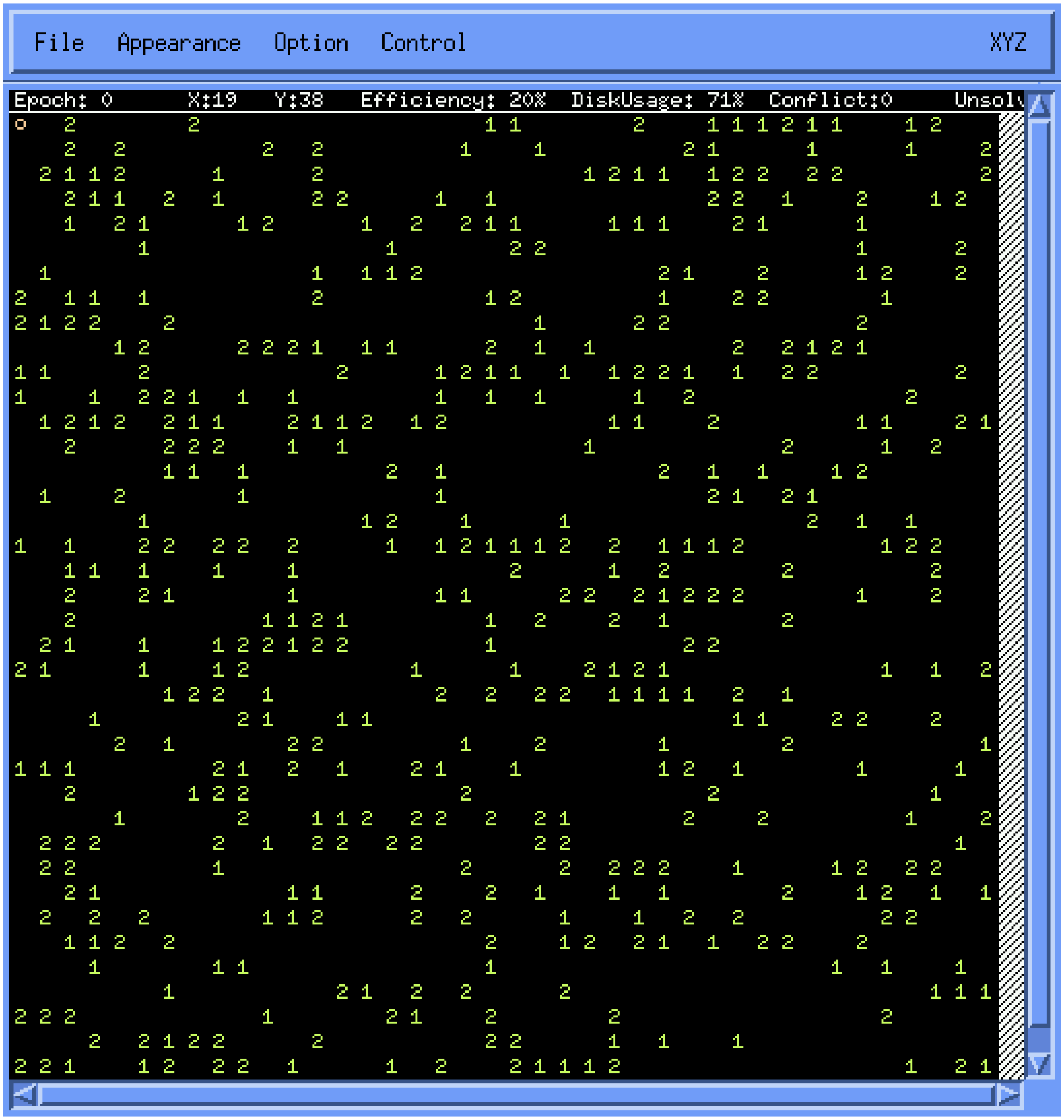


Looking at the CA space once both *cell* and *ctrl* have been selected, it should look like Figure 1.

Notice the values of 1 and 2 scattered on the CA space. This has been achieved with the randomization tool when creating the *.tmpl*. For more information on the randomization tool, review the *.tmpl* section in the Tools. The values 1 and 2 are located in the *cell* field. It can be thought of as a layer. And the *ctrl* field is a layer as well. Both layers are on the same plane. You can view the different layers (fields) with the field control by selecting or unselecting the options you want to see in the CA space. Although a layer (field) may be unselected, it will still be recognized by the rule when it is run after compiling. It is not visible in the CA space, but it is still used by the rule. For now, unselect the *cell* option so that the only symbol in the CA space is the **o** in the upper-left hand corner. This tutorial will work with only the *ctrl* field.

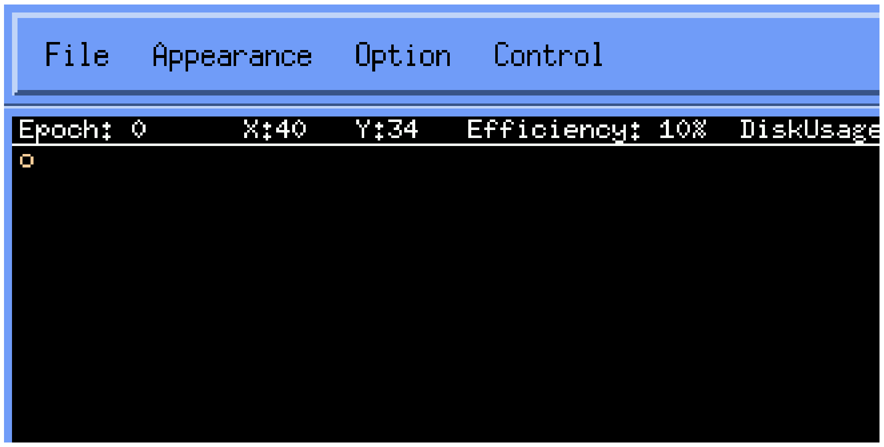
## 2. Examining the Rule

The purpose of this rule is to show the ability of TREND to give the appearance of movement in a one dimensional manner on the CA space. In this example, a symbol will start in the upper left hand corner of the screen and will spread to the right until the entire top line is filled with the symbol. Then it will start to spread downward on the screen again until the entire screen in filled with the symbol. The simulation results are shown in Figure 2, Figure 3, and Figure 4.



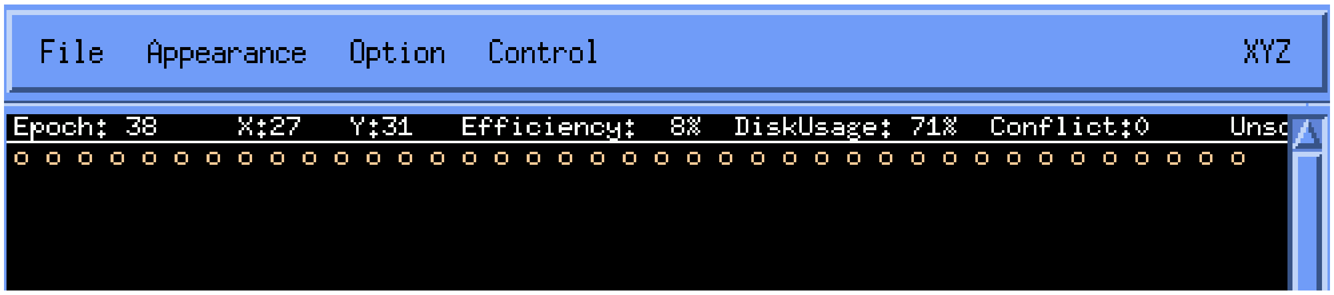
```
Epoch: 0      X:19    Y:38    Efficiency: 20%   DiskUsage: 71%   Conflict:0      Unsolv
```

Figure 1.  The work window of Trend



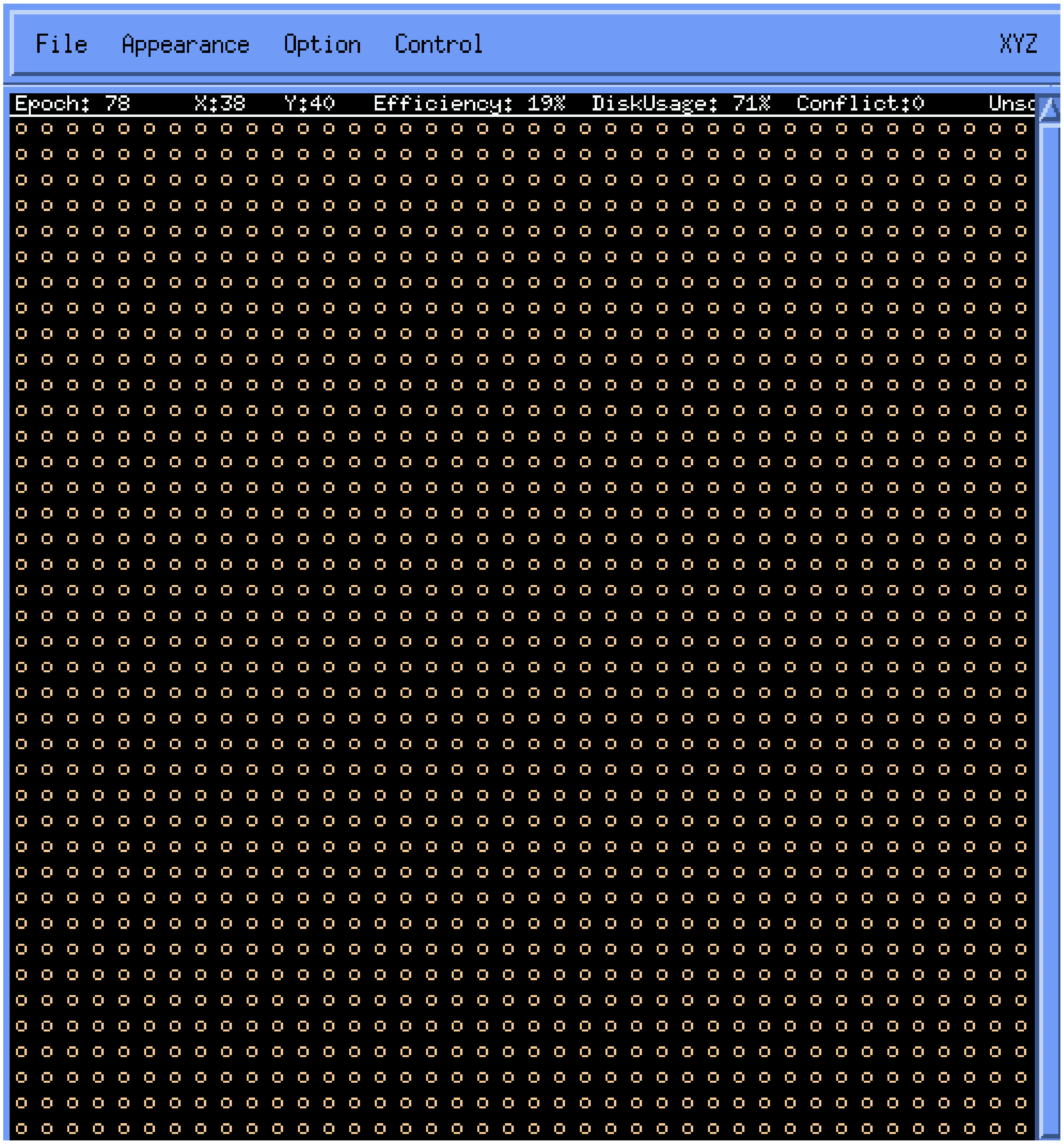
```
Epoch: 0      X:40    Y:34    Efficiency: 10%   DiskUsage
```

Figure 2.  The CA space at epoch 0



```
Epoch: 38      X:27    Y:31    Efficiency:  8%   DiskUsage: 71%   Conflict:0      Unsc
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
```

Figure 3. The  CA space at epoch 38



```
Epoch: 78      X:38    Y:40    Efficiency: 19%   DiskUsage: 71%   Conflict:0      Unsc
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
```

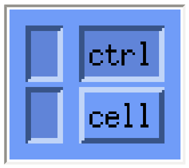
Figure 4. The  CA space at epoch 78

first page. There are several lines with nothing but comments. This is done for easy interpretation and to make it more readable. The comments are shown in bold. Each line is numbered in the table below to allow the code to be looked at line by line.

| line | code |
|---|---|
| 1 | default cell=cell; **// has values of 0,1,2 and .** |
| 2 | default ctrl=ctrl; **// value will be assigned when you choose the symbol** |
| 3 | |
| 4 | /******************************************************************/ |
| 5 | /**************************DECLARATIONS**********************/ |
| 6 | /******************************************************************/ |
| 7 | |
| 8 | int sum; **// initialize integer named "sum"** |
| 9 | int table[]={0, 0, 2, 0, 2, 0, 1}; **//initialize integer named "table"** |
| 10 | |
| 11 | /******************************************************************/ |
| 12 | /**********Code to Perform the Movement From Left to Right*********/ |
| 13 | /******************************************************************/ |
| 14 | |
| 15 | if (ctrl==0 && left:ctrl) **// if CTRL is nothing and LEFT CTRL is "alive"** |
| 16 | ctrl=left:ctrl-1; **//assign CTRL the value LEFT CTRL -1** |
| 17 | |
| 18 | else if (no:ctrl==down:ctrl && ctrl>1) **// if the TOP CTRL is equal to DOWN CTRL & CTRL>1** |
| 19 | ctrl--; **// only reduce by 1 if greater than 1** |
| 20 | |
| 21 | /******************************************************************/ |
| 22 | /***********Code to Perform the Movement Down the Screen**********/ |
| 23 | /******************************************************************/ |
| 24 | |

| | |
|---|---|
| 25 | if (no:ctrl==top:ctrl+1 &#124;&#124; no:ctrl==0 && top:ctrl==225) |
| 26 | { |
| 27 | ctrl=no:ctrl+1; |
| 28 | sum=nw:cell+no:cell+ne:cell; |
| 29 | cell=table[sum]; |
| 30 | } |
| 31 | **/*If NO CTRL is equal to TOP CTRL plus 1, then CTRL is assigned the** |
| 32 | **value of NO CTRL plus 1 and sum is assigned the value of NW:CELL + NO:CELL + NE:CELL and cell** |
| 33 | **is assigned the value of the array sum.*/** |

The first two lines are the initialization of the fields *Cell* and *ctrl* are the fields in this example.

| | | |
|---|---|---|
| 1 | default cell=cell; **// has values of 0,1,2 and .** | |
| 2 | default ctrl=ctrl; **// value will be assigned when you choose the symbol** |  |

Lines one and two basically say that a field (either *cell* or *ctrl*) should keep it's value if there are no statements that affect the value according to the rule.

Lines four, five and six are comments to show that the next portion of the code are going to be related to the declarations.

| | |
|---|---|
| 4 | **/*************************************************************/** |
| 5 | **/*************DECLARATIONS*******************************/** |
| 6 | **/*************************************************************/** |

Line eight is declaring an integer with the name "sum". This integer is used later in the code. It could have been declared just prior to it's use, but it is common to group all the declarations.

| | |
|---|---|
| 8 | int sum; **// initialize integer named "sum"** |

Line nine is also a declaration. It is declaring an integer named "table". This is an array that is used later in the code.

Lines one and two basically say that a field (either *cell* or *ctrl*) should keep it's value if there are no statements that affect the value according to the rule.

Lines four, five and six are comments to show that the next portion of the code are going to be related to the declarations.

```
4 /*************************************************************/
5 /************DECLARATIONS******************************/
6 /*************************************************************/
```

Line eight is declaring an integer with the name "sum". This integer is used later in the code. It could have been declared just prior to it's use, but it is common to group all the declarations.

```
8 int sum;          // initialize integer named "sum"
```

Line nine is also a declaration. It is declaring an integer named "table". This is an array that is used later in the code.

```
9 int table[]={0, 0, 2, 0, 2, 0, 1};    //initialize integer named "table"
```

The next three lines are again just comments to show the next portion of code are going to be related to a particular action.

```
11 /***********************************************************/
12 /******Code to Perform the Movement From Left to Right*********/
13 /***********************************************************/
```

The lines below express a compound condition for the *ctrl* fields. If this field has a *ctrl* value of 0 (zero), and the *ctrl* field to the left has a value of something other than 0, then assign the value of the *ctrl* to the left to the current *ctrl* less one. Basically saying if this one is "dead", and the one to it's left is "alive", make this one "alive" with a value of the one to the left minus one.

```
15 if (ctrl==0 && left:ctrl)      // if CTRL is nothing and LEFT CTRL is "alive"
16      ctrl=left:ctrl-1;       //assign CTRL the value LEFT CTRL -1
```

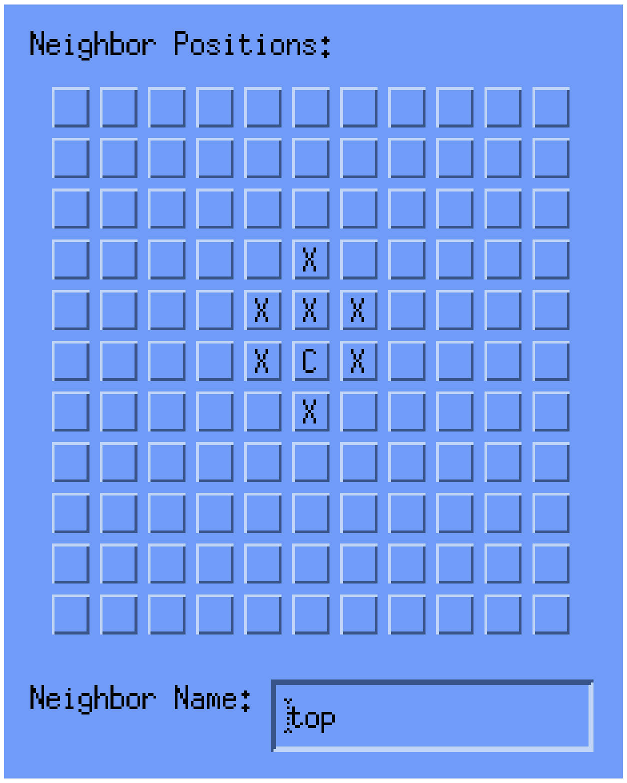
```
18 else if (no:ctrl==down:ctrl && ctrl>1)
19      ctrl--;
```

|  |  |
|---|---|
| 15 | if (ctrl==0 && left:ctrl)       **// if CTRL is nothing and LEFT CTRL is "alive"** |
| 16 |       ctrl=left:ctrl-1;        **//assign CTRL the value LEFT CTRL -1** |

**2. Examining the Rule** *(continued)*

|  |  |
|---|---|
| 18 | else if (no:ctrl==down:ctrl && ctrl>1) |
| 19 |       ctrl--; |

Lines 18 and 19 are the second option of the IF statement when the first part is not proven true. Line 18 says that if the **NO:CTRL equals** the **DOWN:CTRL**, and **CE** (shown with a *C* in the table below) has a value greater than 1, then execute the statement in line 19. The **TOP:CTRL** is shown in the image below. It is the top neighbor (shown with an X). Below the image is a table that explains the neighbors for this rule.



Neighbor Positions:

Neighbor Name: top

In this rule, each field has 7 neighbors. In the CA space window, click on **File ->> Modify Template.** Look at the portion of the window that appears labled *Neighbor Positions*. The **C** is considered a particular field and the **X** are the neighbors of that particular field. The field directly above the **C** is called the **NO:CTRL**. The field directly above the **NO:CTRL** is called **TOP:CTRL**. The names are user defined. Below is a table showing the names of each of the neighbors in this rule.

|  |  |  |
|---|---|---|
|  | TOP:CTRL |  |
| NE:CTRL | NO:CTRL | NW:CTRL |
| EA:CTRL | CE | WE:CTRL |

SO:CTRL

## 2. Examining the Rule

Lines 21 - 23 are again just comments to show the next set of lines are related to a particular action.

```
21 /***********************************************************/
22 /*********Code to Perform the Movement Down the Screen*********/
23 /***********************************************************/
```

Lines 25 - 30 affect the appearance of the path moving down the screen.

```
25 if (no:ctrl==top:ctrl+1 || no:ctrl==0 && top:ctrl==255)
```

Line 25 states that if the NO:CTRL is equal to the value of the TOP:CTRL, **OR** the NO:CTRL has a value of 0 (zero) and the TOP:CTRL has a value of 255, then execute the statement that follows.

Line 26 is just an **{** (open brace) that allows more than one line to be included in the statement. Just as it does in C++.
Line 27 says to make the particular CTRL that is applicable to the rule equal to it's NO:CTRL +1.

```
27     ctrl=no:ctrl+1;
```

Line 28 takes the 3 neighbors just above the particular CTRL and adds their values together to get a *sum*. That *sum* is an integer declared earlier in line 8. Sum is used in line 29. CELL is assigned the value of the value found in the table for *sum*. Line 30 is a close brace to show the end of the statements grouped under the "if" condition in line 25.

```
25 if (no:ctrl==top:ctrl+1 || no:ctrl==0 && top:ctrl==255)
26     {
27     ctrl=no:ctrl+1;
28     sum=nw:cell+no:cell+ne:cell;
29     cell=table[sum];
30     }
31 /*If NO CTRL is equal to TOP CTRL plus 1, then CTRL is assigned the value
   of NO CTRL plus 1 and sum is
```

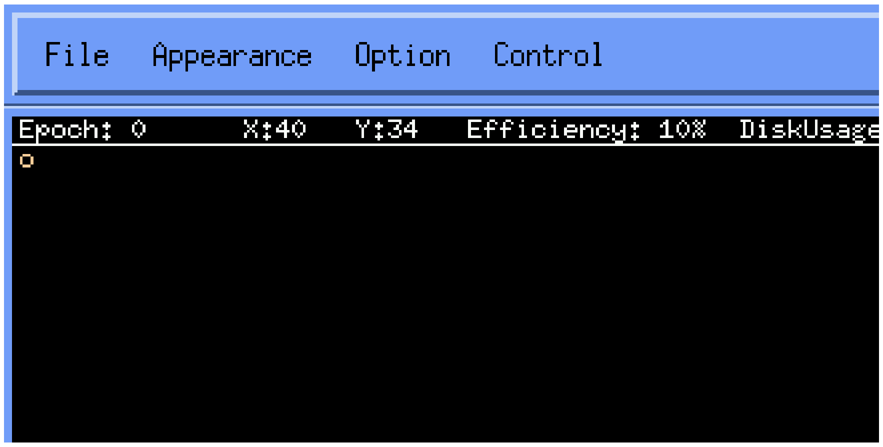
| | |
|---|---|
| 32 | assigned the value of NW:CELL + NO:CELL + NE:CELL and cell is assigned the value of the array sum. */ |

Lines 31 and 32 are just comments to complete the rule.

## 3. The Rule in Action

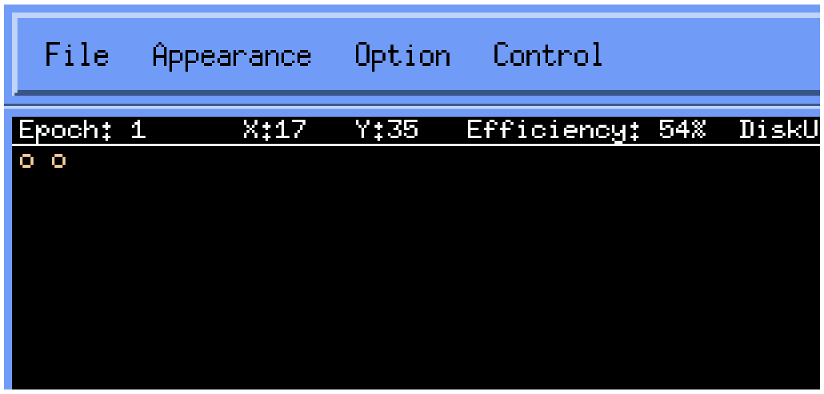
The previous pages have disected the rule in order to explain each line. Below is an example of the rule in action.

After loading the rule, the CA space should look like this at Epoch 0 (stage 0).



```
Epoch: 0      X:40   Y:34   Efficiency: 10%  DiskUsage
o
```

Next, compile the rule. Once it is compiled successfully, use the controls to increase the Epoch number. Below is what Epoch 1 should look like.

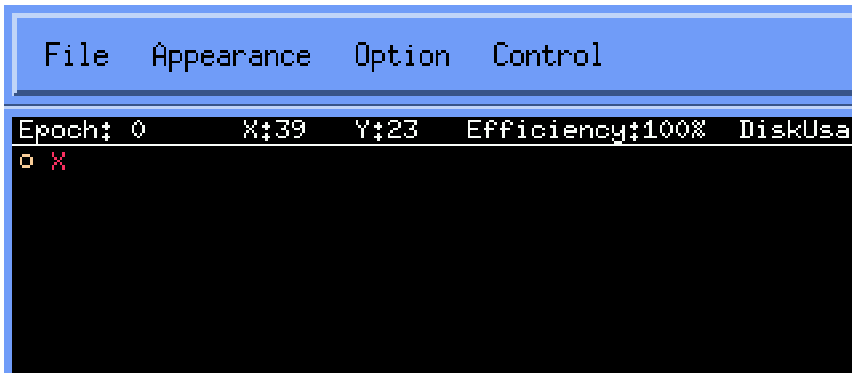


```
Epoch: 1      X:17   Y:35   Efficiency: 54%  DiskU
o o
```

In Epoch 1, the symbol **o** appears to have spread to the right. There are now two symbols in the CA space. This was caused by lines 15 and 16 of the rule being applied to the original symbol in the CA space.

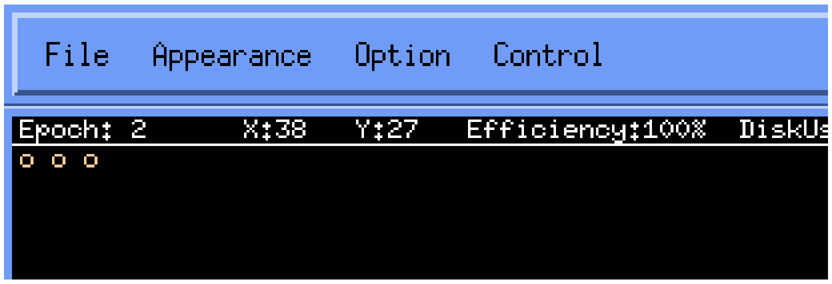
| | |
|---|---|
| 15 | if (ctrl==0 && left:ctrl)  **// if CTRL is nothing and LEFT CTRL is "alive"** |
| 16 | ctrl=left:ctrl-1;  **//assign CTRL the value LEFT CTRL -1** |

Going back to epoch 0, in the image below the red X has been placed in the CA space to show the particular cell we will be applying lines 15 and 16 to. This X is just to show the position, and has nothing to do with the rule itself.

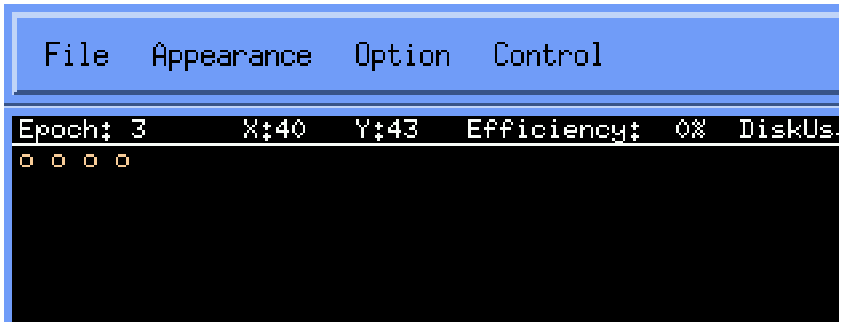


```
Epoch: 0       X:39   Y:23   Efficiency:100%   DiskUsa
o X
```

Looking at lines 15 and 16, the position with the red X has a value of zero for the field ctrl. And the position to the left of it has a value of something other than zero. This satifies the condition in line 15, so line 16 is to be executed when moving to Epoch 1. That is shown in the image above with the two **o** symbols in the CA space. The rule is causing the appearance of the symbols spreading to the right.



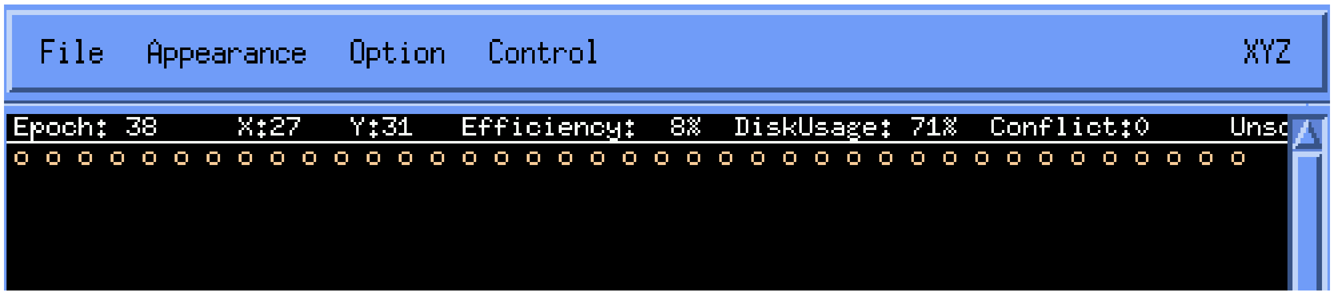
```
Epoch: 2       X:38   Y:27   Efficiency:100%   DiskUs
o o o
```



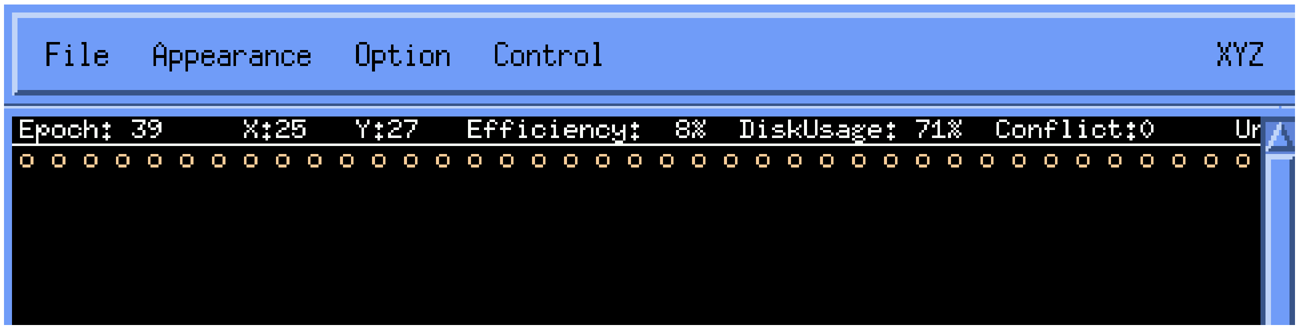
```
Epoch: 3       X:40   Y:43   Efficiency:  0%   DiskUs
o o o o
```

The images above show that the spreading continues as the number of epochs increases. This will continue until the symbols reach the right-hand side of the screen. At that point, the lines 15 and 16 no longer apply to the symbols in the CA space.

The symbols continue to spread to the right of the CA space until it reached another **o** symbol in the CA space. It may be misleading in the example, but the reason the spreading to the right stops is because the condition in line 15 no longer applies. Not because the symbols have reached the edge of the screen. In TREND, there is no edge to the CA space. The symbols continue through to the other side without any break in the line.
The images below show the CA space at epoch 38 and 39.



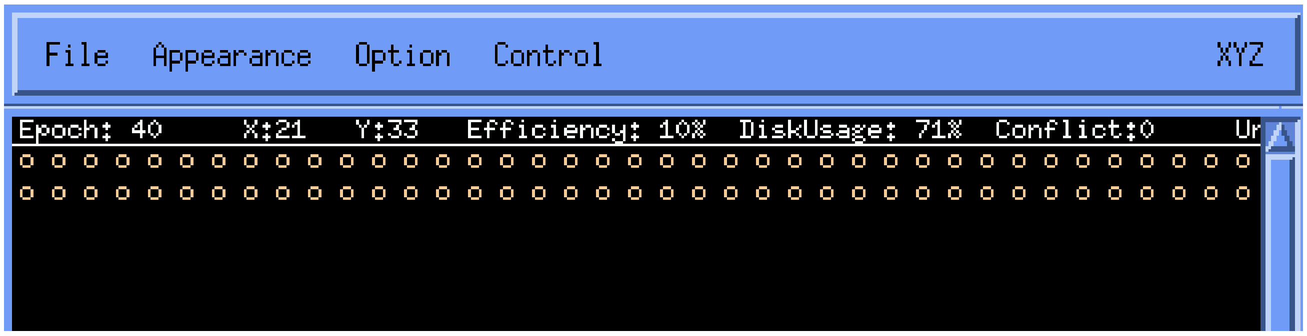
```
Epoch: 38      X:27    Y:31    Efficiency:  8%  DiskUsage: 71%   Conflict:0      Unsc
```



```
Epoch: 39      X:25    Y:27    Efficiency:  8%  DiskUsage: 71%   Conflict:0      Ur
```

At epoch 39, there are no more "dead" *ctrl* fields in the first row. So the condition in line 15 is no longer satisfied. So the statement in line 16 will no longer be executed.

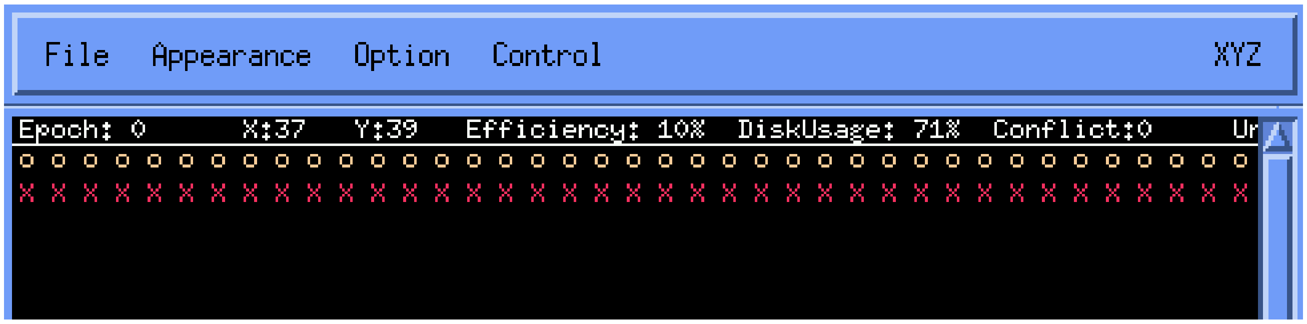
So what happens at the next epoch?



```
Epoch: 40      X:21    Y:33    Efficiency: 10%  DiskUsage: 71%   Conflict:0      Ur
```

At epoch 40, the symbols appear to move down the CA screen. The condition in **bold** in line 25 is now applicable at epoch 40.

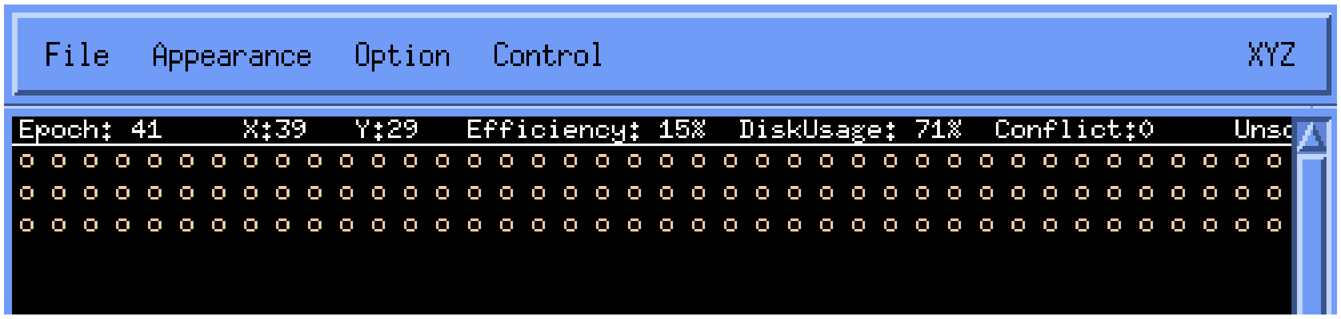
| | |
|---|---|
| 25 | **if (no:ctrl==top:ctrl+1** \|\| no:ctrl==0 && top:ctrl==225) |
| 26 | { |
| 27 | ctrl=no:ctrl+1; |
| 28 | sum=nw:cell+no:cell+ne:cell; |
| 29 | cell=table[sum]; |
| 30 | } |



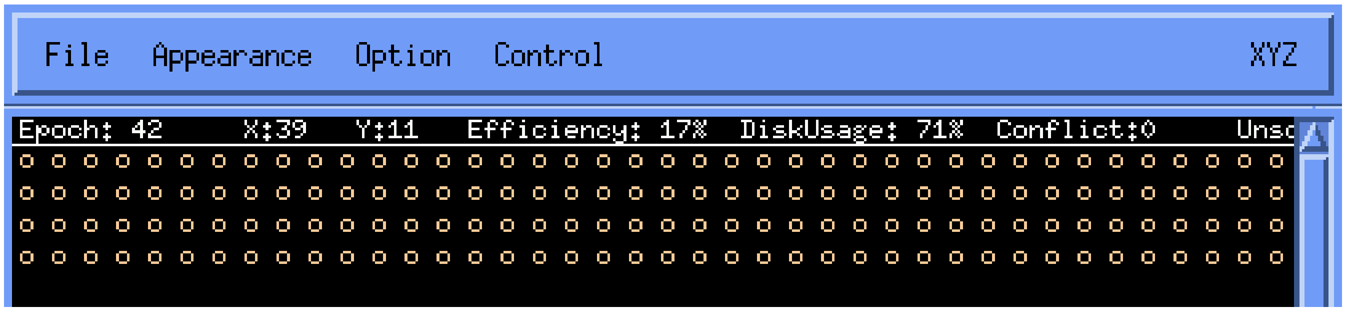
```
Epoch: 0       X:37    Y:39    Efficiency: 10%   DiskUsage: 71%   Conflict:0      Ur
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X
```

In the image above, the red X's are again just to show the fields that are being affected by the condition. Line 25 basically says that if the NO:CTRL has a value of one greater than the TOP:CTRL, then execute line 27 and make the current CTRL the same value as the NO:CTRL plus one.

At epoch 41, the trend continues and the symbols appear to move down the CA screen. This continues until the symbols spread down to the bottom part of the CA screen.



```
Epoch: 41      X:39    Y:29    Efficiency: 15%   DiskUsage: 71%   Conflict:0      Unsc
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
```

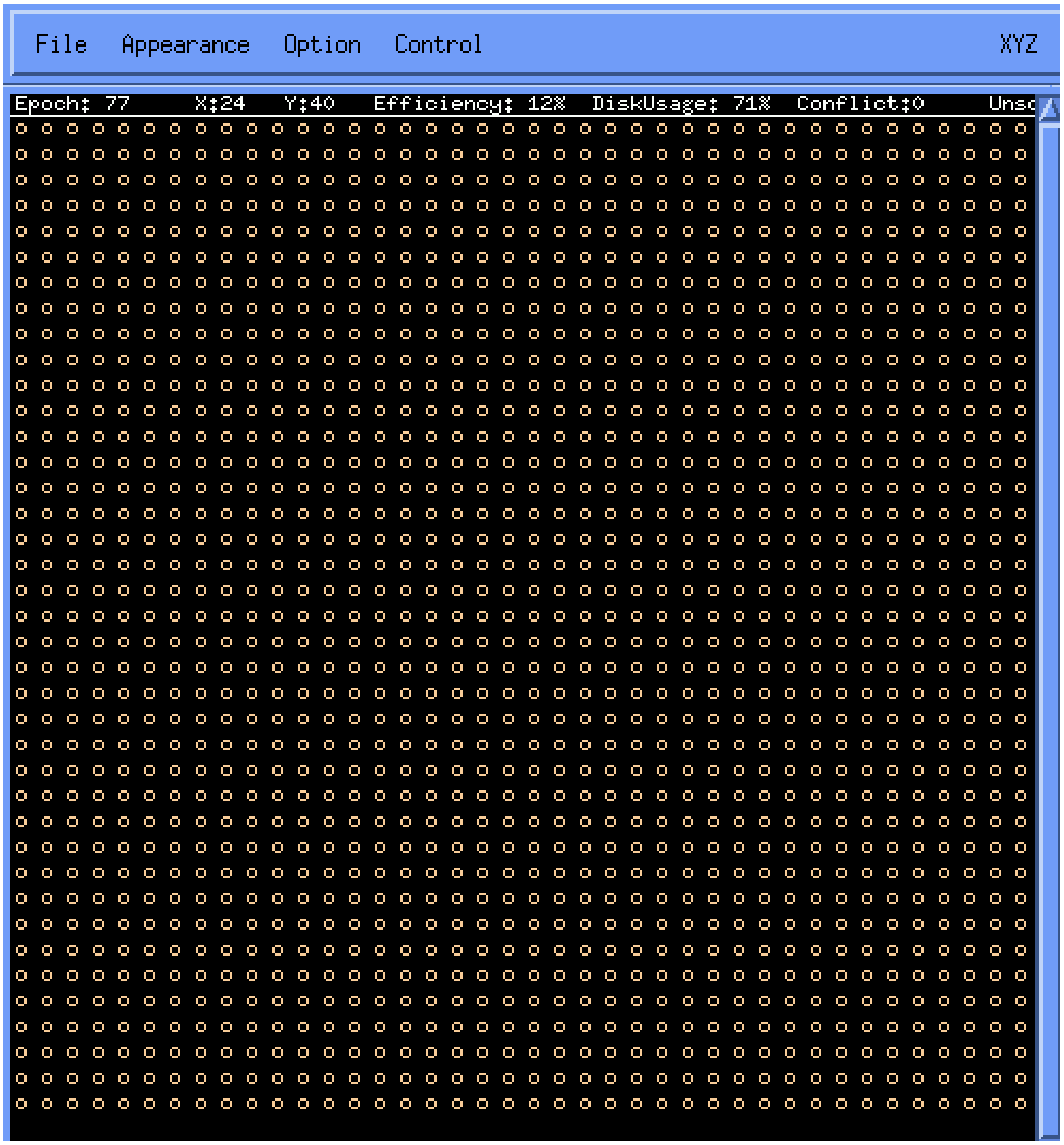


```
Epoch: 42      X:39    Y:11    Efficiency: 17%   DiskUsage: 71%   Conflict:0      Unsc
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
```

At epoch 77, the whole CA space is almost full with the **o** symbol (Figure 5).

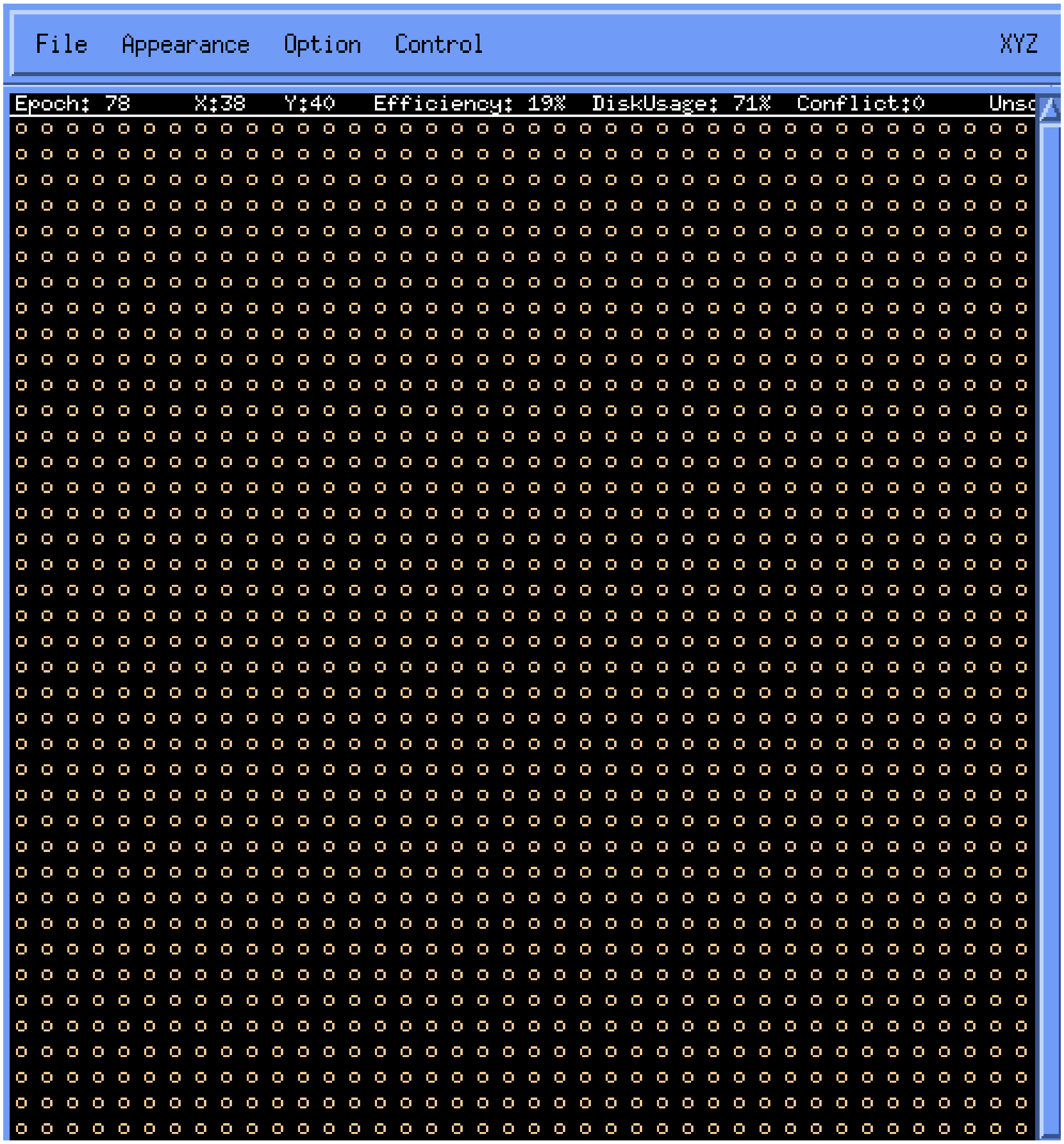
At epoch 78, shown in Figure 6, the entire CA space is full. This means all the *ctrl* cells are "live", so the condition in line 25 no longer is applicable.

At epoch 79, the screen will remain full. The CA screen appears to not change after epoch 78. In fact, the lines are just being recreated over the original lines across the screen going down. It is an infinite loop and will continue to look the same as the epoch number continues to increase.



```
Epoch: 77     X:24    Y:40    Efficiency: 12%   DiskUsage: 71%   Conflict:0      Unsc
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o o
```

Figure 5. The CA space at epoch 77. It is almost full of symbol 'o'



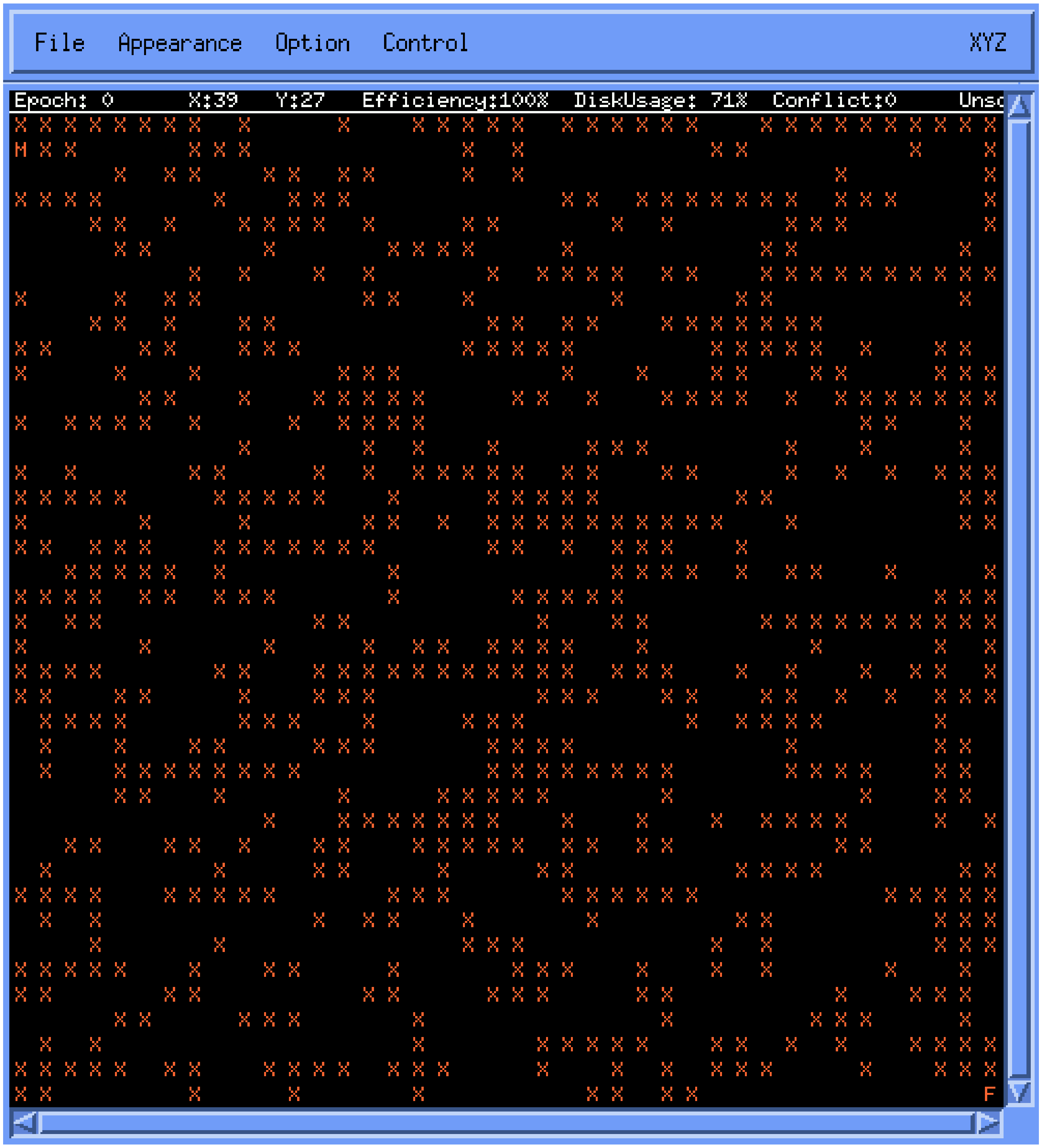
Epoch: 78 X:38 Y:40 Efficiency: 19% DiskUsage: 71% Conflict:0 Unsc

Figure 6. The CA space at epoch 78. The simulation is finished.

# Mouse Maze

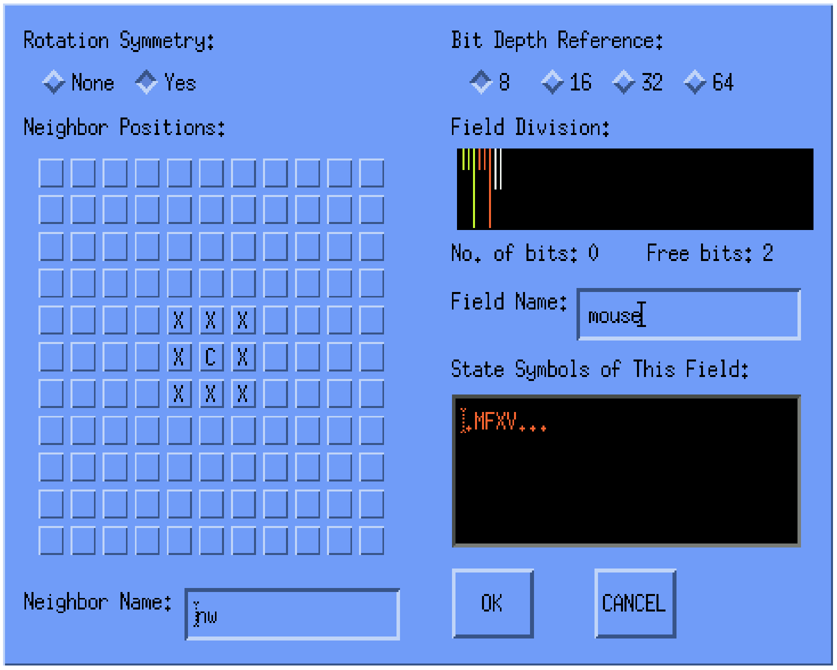
## 1. Mouse Searching For Food

The next example project to look at is the Mouse Maze. The idea is to have a cell start out being considered the mouse. And that mouse has to go through the maze until it finds food. Once the mouse is able to find food, the program stops. In this example, there is only one mouse and only one cell that is the food. Below is the CA space for this example. Again this can be loaded by finding the local directory in which the TREND is located. Select the **File** option from the .CA screen and select **Get template**... Navigate to the local directory in which the project has been saved. Next, find and select the project named **hhmouse.tmpl**. TREND will open the .rule and .ca screens as well. This is the case only as long as they all exist in the same directory



```
Epoch: 0    X:39    Y:27    Efficiency:100%    DiskUsage: 71%    Conflict:0    Unsc
```

The mouse is assigned the symbol "M", and is in the upper left hand corner of the CA screen. The food is assigned the symbol "F" and is in the lower right hand corner. The cells assigned the symbols "X" are the walls of the maze. The mouse must go around the walls in order to find the food. The path the mouse takes will be represented by the symbols "^, V, >, <". And the symbol "d" is used to represent a dead end. And the symbol "V" is assigned to the mouse when it finds the food. Below are the *.tmpl* for the two fields in this example.



Rotation Symmetry: None Yes

Bit Depth Reference: 8 16 32 64

Neighbor Positions:

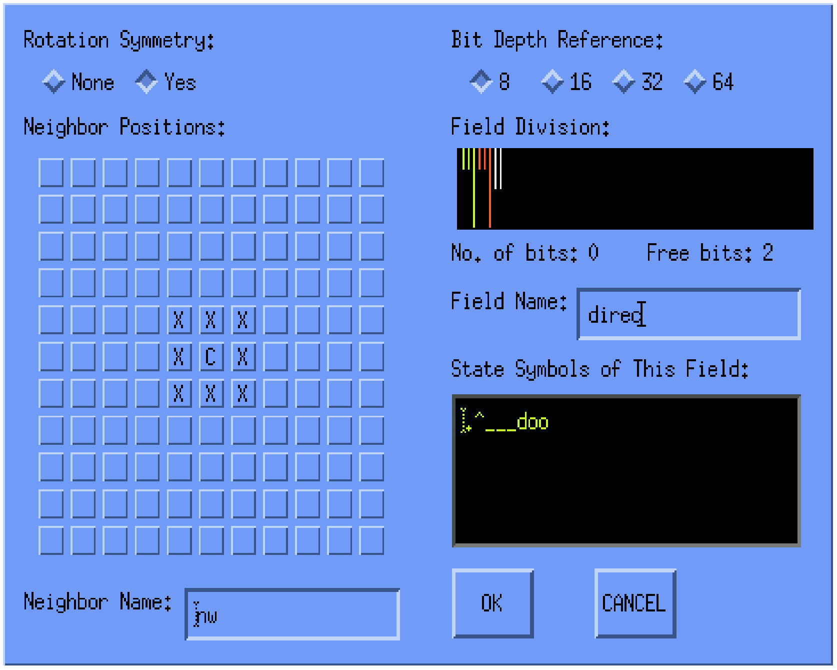
Field Division:

No. of bits: 0 Free bits: 2

Field Name: mouse

State Symbols of This Field: MFXV...

Neighbor Name: nw



Rotation Symmetry: None Yes

Bit Depth Reference: 8 16 32 64

Neighbor Positions:

Field Division:

No. of bits: 0 Free bits: 2

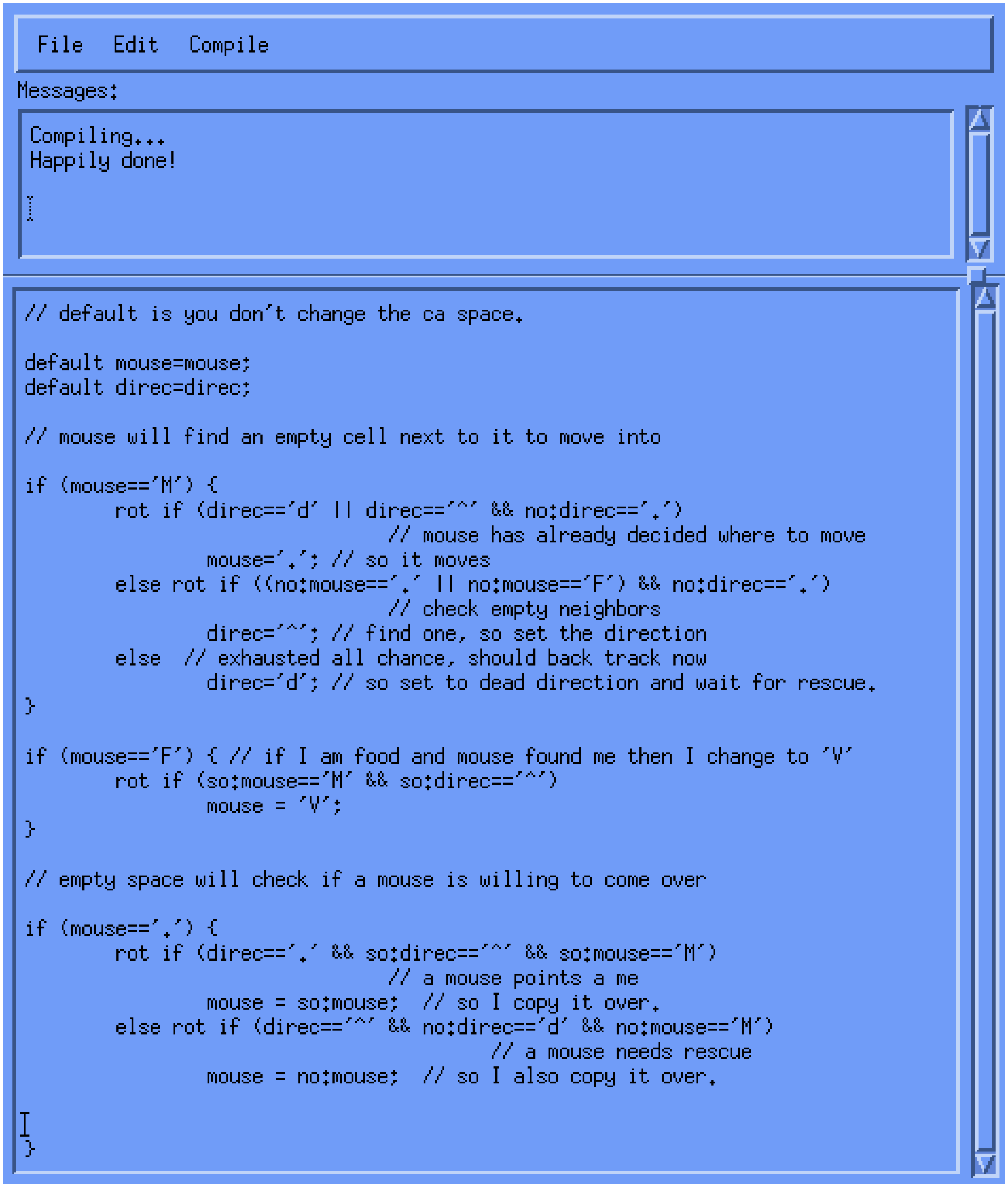
Field Name: direc

State Symbols of This Field: ^___doo

Neighbor Name: nw

## 2. Mouse Maze Code

The code below is the rule for the Mouse Maze project. As with the other examples, there are several lines of code that have comments. It is recommended to make comments often in the code to help debugging and maintenance of the code.



```
Compiling...
Happily done!
```

```
// default is you don't change the ca space.

default mouse=mouse;
default direc=direc;

// mouse will find an empty cell next to it to move into

if (mouse=='M') {
        rot if (direc=='d' || direc=='^' && no:direc=='.')
                                // mouse has already decided where to move
                mouse='.'; // so it moves
        else rot if ((no:mouse=='.' || no:mouse=='F') && no:direc=='.')
                                // check empty neighbors
                direc='^'; // find one, so set the direction
        else  // exhausted all chance, should back track now
                direc='d'; // so set to dead direction and wait for rescue.
}

if (mouse=='F') { // if I am food and mouse found me then I change to 'V'
        rot if (so:mouse=='M' && so:direc=='^')
                mouse = 'V';
}

// empty space will check if a mouse is willing to come over

if (mouse=='.') {
        rot if (direc=='.' && so:direc=='^' && so:mouse=='M')
                                // a mouse points a me
                mouse = so:mouse;  // so I copy it over.
        else rot if (direc=='^' && no:direc=='d' && no:mouse=='M')
                                     // a mouse needs rescue
                mouse = no:mouse;  // so I also copy it over.

}
```

Below is a table showing the line numbers of the code shown above. This will help in disecting the code line by line.

| **line** | **code** |
|---|---|
| 1 | // default is you don't change the ca space |
| 2 | |
| 3 | default mouse=mouse; |
| 4 | default direc=direc; |
| 5 | |
| 6 | // mouse will find empty cell next to it to move into |
| 7 | |
| 8 | if (mouse=='M') { |
| 9 |     rot if (direc=='d' \|\| direc=='^' && no:direc=='.') |
| 10 |             // mouse has already decided where to move |
| 11 |         mouse='.'; // so it moves |
| 12 |     else rot if ((no:mouse=='.' \|\| no:mouse=='F') && no:direc=='.') |
| 13 |             // check empty neighbors |
| 14 |         direc='^'; // find one, so set the direction |
| 15 |     else // exhausted all chance, should back track now |
| 16 |         direc='d'; // so set to dead direction and wait for rescue |
| 17 | } |
| 18 | |
| 19 | if (mouse=='F') { //if I am food and mouse found me then I change to 'V' |
| 20 |     rot if (so:mouse=='M' && so:direc=='^') |
| 21 |         mouse= 'V'; |
| 22 | } |
| 23 | |
| 24 | // empty space will check if a mouse is willing to come over |
| 25 | |
| 26 | if (mouse=='.') { |
| 27 |     rot if (direc=='.' && so:direc=='^' && so:mouse=='M') |
| 28 |             // a mouse points at me |

```
29          mouse=so:mouse; //so I copy it over
30      else rot if (direc=='.' && no:direc=='d' && no:mouse=='M')
31              // a mouse needs rescue
32          mouse=no:mouse; // so I also copy it over
33  }
```

### 3. Explanations of Code

Line one of the code is just a comment declaring the default action of the code. If a cell doesn't fit a condition in the code, it is left alone (not changed).

```
1 // default is you don't change the ca space
```

Line three and four are declarations. They make the defaults for the condition explained in line one.

```
3 default mouse=mouse;
4 default direc=direc;
```

Line six is another comment that explains the code in line eight.

```
6 // mouse will find empty cell next to it to move into
```

As the comments in line six suggest, line eight will search the CA space for the mouse 'M'. The symbol 'M' is a value in the field labeled "mouse". This can be seen in the screen print of the *.tmpl* on page one. If a symbol 'M' is found in the CA space, the program will continue with the lines of code after line eight until it finds a close brace.

```
8 if (mouse=='M') {
```

Line nine is a condition that is started with a "rotated if". A "rotated if" is a short cut in the TREND language. It is a very powerful tool that allows the user to combine 4 sets of conditions into one. For now think of it as a normal "if". Line nine basically says that if any cell in the CA space has a **direc**of 'd', then make the **mouse**value for that cell equal to '.'. Meaning, if the cell has a direc value of 'd' (it is a dead end), then make the mouse value for that cell equal to 0 (zero value is shown with a '.'). There is an in this line. It suggests that if the first part of the condition is not true, then check to see if the second part is true. Check if **direc**is equal to '^' *and* the cell above it has a

**direc**of '.' (zero), then assign the value of mouse to '.' (zero).

```
9    rot if (direc=='d' || direc=='^' && no:direc=='.')
10       // mouse has already decided where to move
11     mouse='.'; // so it moves
```

With the "rotated if" in line nine, the code is shortened significantly. Here is another way of coding the conditions presented in line nine.

```
if (direc=='d' || direc=='^' && no:direc=='.')
    else if (direc=='d' || direc=='>' && ea:direc=='.')
        else if (direc=='d' || direc=='V' && so:direc=='.')
            else (direc=='d' || direc=='<' && we:direc=='.')
```

The "rotated if does the rotation automatically. The rule is looking for the conditions to be true from it's North, South, East, and West neighbors. The "rotated if" will check all of these neighbors for the condition automatically. This saves a lot of time and space when coding.

Line ten is another comment stating that if the condition in line nine is found to be true, the mouse knows which way it is going to go.

```
10       // mouse has already decided where to move
```

Line 11 is the statement to be executed when the condition in line nine is found to be true. Basically it just makes the value of the mouse '.' (zero).

```
11     mouse='.'; // so it moves
```

Line 12 continues the "if" that was started in line eight. Line 12 is the first "else" to the condition in line eight. It continues that if the particular cell has the value of 'M' in the mouse field and the cell in the mouse field just above it has a value of '.' (zero), or has a value of 'F' (food) and a direc value of '.' (zero), then execute the statement that follows the condition.

```
12   else rot if ((no:mouse=='.' || no:mouse=='F') && no:direc=='.')
```

Line 13 is just a comment explaining the condition in line 12. Saying the cell in the mouse field with a value of 'M' is checking it's neighbors.

```
13              // check empty neighbors
14          direc='^'; // find one, so set the direction
```

The statement in line 14 says to assign a value of '^' to the direc field in the cell that meets the condition in line 12. So assign a direction to the neighbor that has a value of '.' for the mouse field or a value of 'F' in the mouse field and '.' in the direc field.

Line 15 is the default for the condition started in line eight. It is the "else" that catches any of the fields that have a value of 'M' in the mouse field, but are not true in the second part of the condition. So if a cell has a value of 'M' in the mouse field, but does not prove true for lines nine and 12, then execute the statement that follows in line 16.

```
15      else // exhausted all chance, should back track now
16          direc='d'; // so set to dead direction and wait for rescue
```

Lines 15 and 16 set the value of direc to 'd', to show it has hit a dead end. The mouse can no longer move anywhere other than where it came from because it has reached a dead end. The value of 'd' for the field direc will prevent the mouse from entering the same cell at a later point. There is no reason for the mouse to enter a cell that is a dead end.

Line 17 is a close brace to finish the "if" condition started in line eight.

```
17      }
```

Lines 19 through 22 are to find whether the mouse has found the food yet.

```
19  if (mouse=='F') { //if I am food and mouse found me then I change to 'V'
20      rot if (so:mouse=='M' && so:direc=='^')
21          mouse= 'V';
22  }
```

Line 19 starts the condition by checking to see if a cell has a value of 'F' in the mouse field. Line 20 goes from there and continues the condition. If a cell has a value of 'F' in the mouse field, then it checks with a "rotated if" whether a neighbor (North, South, East, West) has a value of 'M' in the mouse field and if the neighbor has a value of '^' in the direc field. In other words, if a cell has a value of 'F', it is considered the food. If it is the food, and it has a neighbor that is a mouse, and that mouse is pointing or coming toward the food, then execute the statement. The statement in line 21 says that the cell that is the food is assigned the value of 'V' for victory. Victory is once the food is found by the mouse. Line 22 finishes it with a close brace.

Line 24 is just a comment to explain what the following lines of code will accomplish.

```
24 // empty space will check if a mouse is willing to come over
```

Lines 26 through 33 deal with changing the value of the mouse field to 'M', when it is a neighbor of a cell that is already the mouse. Therefore giving the impression that the mouse is moving. Line 26 starts the condition by stating that the cell in question must have a mouse field with the value of '.' (zero).

```
26 if (mouse=='.') {
```

Line 27 continues the condition with the first part of the "rotated if". It states the direc field for the cell must have a value of '.' and the neighbor cell must have a direc field value of '^', and it's mouse field value must equal 'M'. The interpretation of this line is basically the cell must not have a direc value set, and the neighbor cell must have a direc value that is pointing at the cell and it must be the mouse. So to say it yet another way. The neighbor cell must be the mouse and must be pointing at the cell in question in order for the cell to change into the mouse.

```
27     rot if (direc=='.' && so:direc=='^' && so:mouse=='M')
```

Line 28 is just a comment stating the condition in line 27.

```
28             // a mouse points at me
```

Line 29 is the statement that is to be executed when the conditions in lines 26 and 27 are met.

```
29         mouse=so:mouse; //so I copy it over
```

Assign the value of the neighbor cell that is the mouse, to this cell. Make "me" the mouse.

Line 30 is the "else" of the rotated if. This will capture all others that do not meet the conditions in line 27.

```
30     else rot if (direc=='.' && no:direc=='d' && no:mouse=='M')
```

If the cell in question has a mouse field value of '.' (zero), and a direc value of '.'. But the neighbor has a direc value of 'd' (dead-end) and the neighbor is the mouse, then execute the statement that follows the condition.

Line 31 is a comment that states the mouse needs to be rescued from the dead-end.

```
31              // a mouse needs rescue
```

Line 32 is the statement to be executed when the conditions in line 30 are met. Assign the value of the neighbor cell to this cell. Or in other words, make "me" the mouse. Appears as if you are moving the value of the mouse from the neighbor cell into the cell in question.

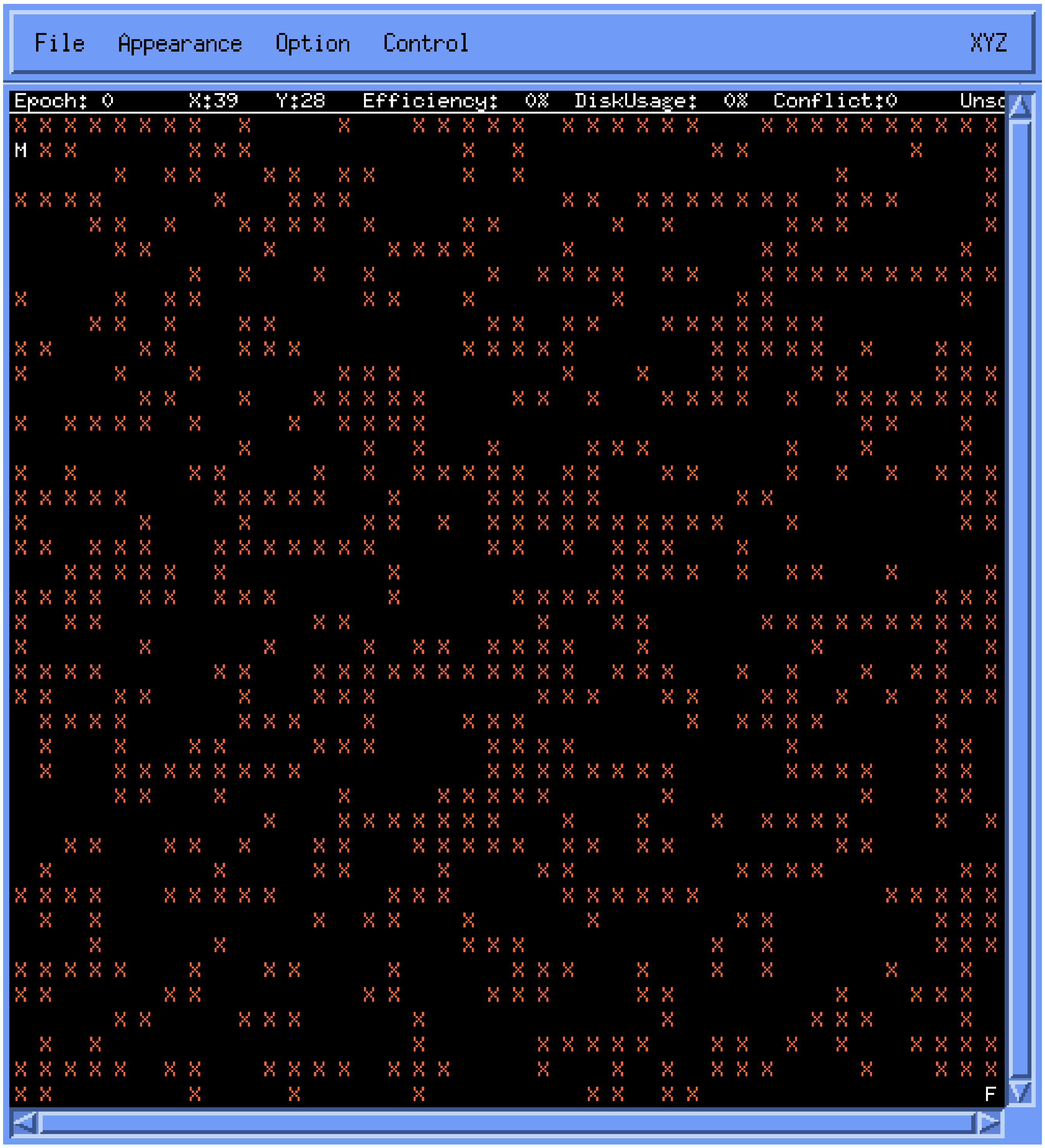
```
32          mouse=no:mouse; // so I also copy it over
33  }
```

And the last line is the close brace to finish the "rotated if" that was started with line 26.

## 4. Simulation Results

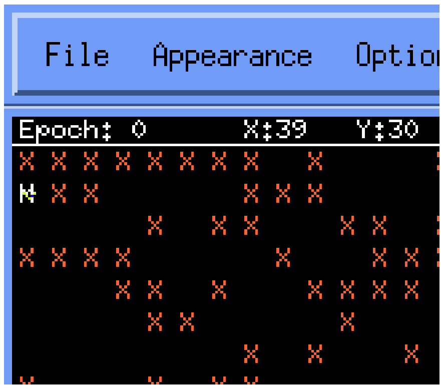
Figure 1 is the CA space for this project. It is the same as the image on page one of this tutorial, except for the difference in color for the 'M' and the 'F'. They have been highlighted in white just to make them easier to see in this tutorial. Again, this is only for the use of the tutorial. The 'M' and 'F' will not appear in this color in the downloaded version of this project.

Notice the epoch is at 0. Once the rule has been compiled and the epoch has been advanced to 1, Figure 2 will appear. Again the color of the mouse and the food will appear in white only for this tutorial.



```
Epoch: 0        X:39    Y:28    Efficiency:  0%   DiskUsage:  0%   Conflict:0      Unsc
```

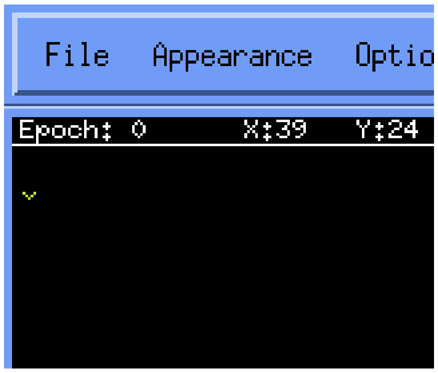
Figure 1.   The CA space of Trend at the beginning of simulation



```
Epoch: 0        X:39    Y:30
```

Figure 2.   The CA space when the mouse move one step

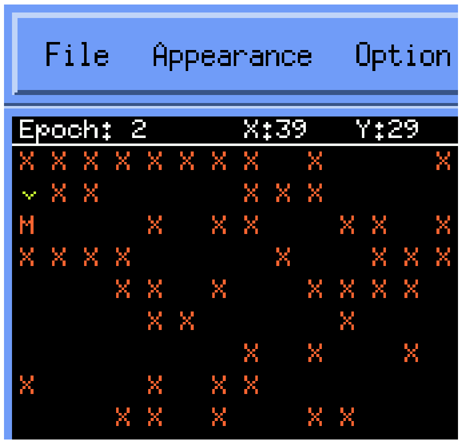
Concentrating on only the section of the CA space that has the mouse in it, notice that behind the 'M' is a down-arrow 'V'. That is the direction field. To see it more clearly, use the fields tool and unclick the mouse fields in order to only see the direc fields.



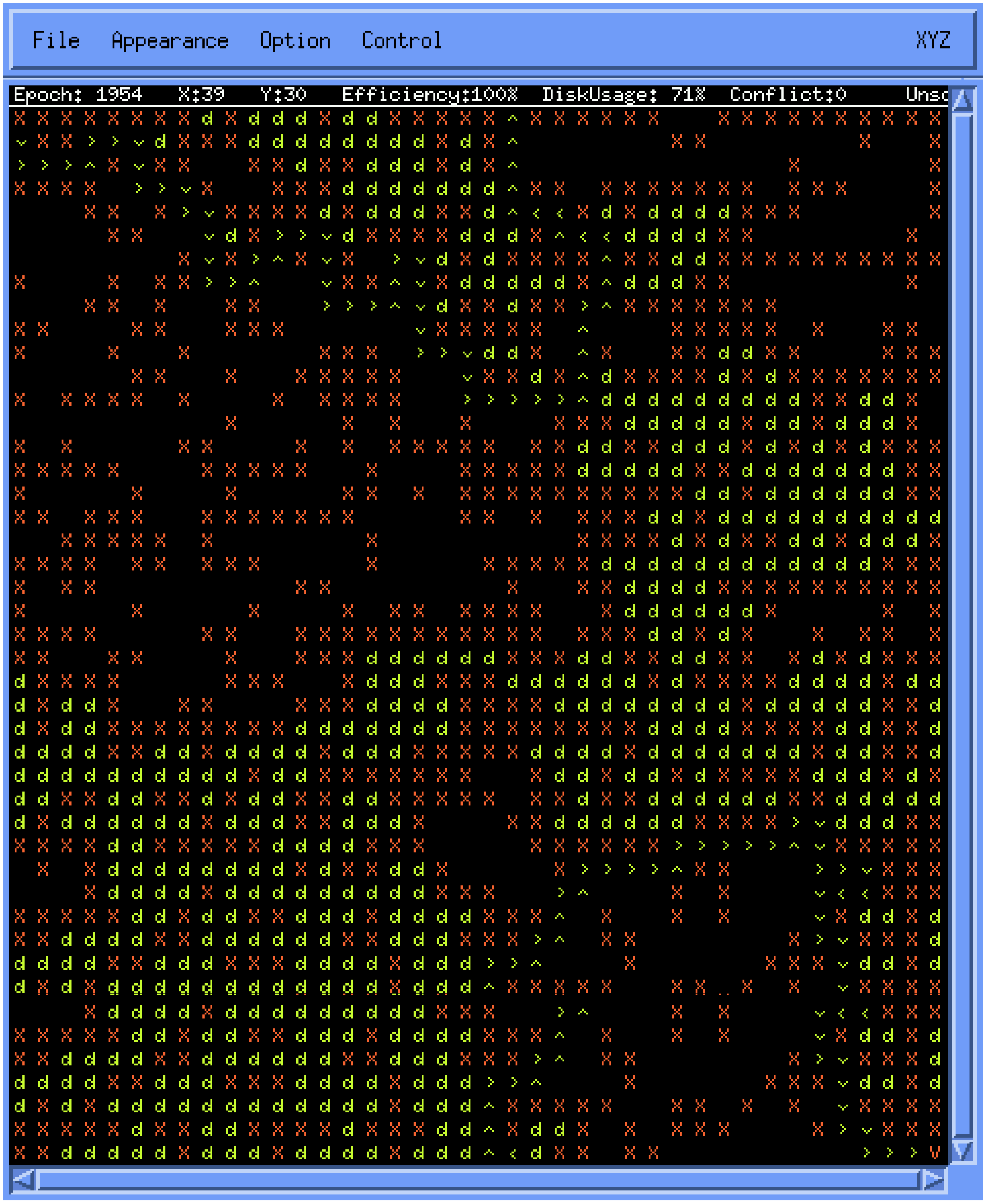
Return to the previous screen by enabling the mouse fields again with the fields tool. The direc field is showing a down-arrow. That is due to the rule, specifically lines 12 and 14. Line 12 starts with the rotated if and is looking for a neighbor that has a mouse value of '.' and a direc value of '.'. That is exactly what the cell below the mouse is. It has a zero value for the mouse field and a zero value for the direc field. It is considered a dead cell. So the mouse sees that it can go in that direction. But before it assigns that cell the value of 'M', it sets it's own direc value to '^'.

| | |
|---|---|
| 12 | else rot if ((no:mouse=='.' \|\| no:mouse=='F') && no:direc=='.') |
| 13 | // check empty neighbors |
| 14 | direc='^'; // find one, so set the direction |

Advance the epoch to 2 and it will show the mouse appears to move to the cell below it. Actually it doesn't move. But the cell that was the mouse, now has a value of '.' for the mouse field and the cell below it now has a value of 'M'. They traded values as a result of the rule.



Using the fast forward controls, let the program run and watch as the mouse searches for the food and once it finds it, the program stops with the mouse changing to the value of 'V' for victory. The green trail shows the path the mouse has taken and the values of 'd' show any dead-ends the mouse may have encountered along the way.



```
Epoch: 1954   X:39   Y:30   Efficiency:100%  DiskUsage: 71%  Conflict:0     Unsc
X X X X X X X X d X d d d X d d X X X X X ^ X X X X X X     X X X X X X X X X X
v X X > > v d X X X d d d d d d d d X d X ^             X X             X     X
> > > ^ X v X X     X X d X X d d d X d X ^                       X           X
X X X X   > > v X     X X X d d d d d d d ^ X X   X X X X X X X   X X X       X
      X X   X > v X X X X d X d d d X X d ^ < < X d X d d d d X X X           X
        X X     v d X > > v d X X X X d d d X ^ < < d d d d X X             X
              X v X > ^ X v X   > v d X d X X X X ^ X X d d X X X X X X X X X X
X       X   X X > > ^     v X X ^ v X d d d d d X ^ d d d X X               X
      X X   X     X X     > > > ^ v d X X d X X > ^ X X X X X X X
X X       X X     X X X           v X X X X X   ^       X X X X X   X     X X
X       X     X           X X X   > > v d d X   ^ X     X X d d X X       X X X
          X X     X     X X X X X     v X X d X ^ d X X X X d X d X X X X X X X
X   X X X X   X       X   X X X X     > > > > > ^ d d d d d d d d d X X d d X
                  X         X   X     X       X X X d d d d d X d d X d d d X
X   X         X X       X   X   X X X X X   X X d d X X d d d X d X d X d X X X
X X X X X       X X X X X     X       X X X X X d d d d d X X d d d d d d d X X
X         X       X         X X   X   X X X X X X X X X X d d X d d d d d d X X
X X   X X X     X X X X X X X         X X   X   X X X d d X d d d d d d d d d d
    X X X X X   X             X                 X X X X d X d X X d d X d d d X
X X X X   X X   X X X         X         X X X X X d d d d d d d d d d d d X X X
X   X X                 X X               X     X X d d d d X X X X X X X X X X
X         X         X       X   X X   X X X X     X d d d d d d X         X   X
X X X X         X X     X X X X X X X X X X X   X X X d d X d X     X   X X   X
X X     X X       X     X X X d d d d d d X X X d d X X d d X X   X d X d X X X
d X X X X         X X X     X d d d X X X d d d d d d X d X X X X d d d d X d d
d X d d X     X X       X X X d d d d X X X X d d d d d d d d X d d d d d X X d
d X d d X X X X X X X X d d d d d d d X X X X X X X X d d d d X X X X d d X X d
d d d d X X d d X d d d d X d d d X X X X X d d d d X d d d d d d d X d d X X d
d d d d d d d d d d X d d X X X X X X X     X d d X d d X d X X X X d d d X d X
d d X X d d X X d X d d X X d d X X X X X   X X d X X d d d d d d X X d d d d d
d X d d d d d d X d d d X X d d d X       X X d d d d d d X X X X > v d d d X X
X X X X d d X X X X X d d d d X X X         X X X X X X > > > > > ^ v X X X X X
  X   X d d d d d d d d X d X X d d X         X > > > > ^ X X       > > v X X X
      X d d d d X d d d d d d d d d X X X     > ^       X   X       v < < X X X
X X X X X d d X d d X X d d d X d d d d X X X ^   X     X   X       v X d d X d
X X d d d d X X d d d d d d X X d d d X X X > ^   X X             X > v X X X d
d d d d X X d d d X X X d d d d X d d d > > ^       X           X X X v d d X d
d X d X d d d d d d d d d d d d X d d d ^ X X X X X     X X .. X   X   v X X X X
      X d d d d X d d d d d d d d d X X X     > ^       X   X       v < < X X X
X X X X X d d X d d X X d d d X d d d d X X X ^   X     X   X       v X d d X d
X X d d d d X X d d d d d d X X d d d X X X > ^   X X             X > v X X X d
d d d d X X d d d X X X d d d d X d d d > > ^       X           X X X v d d X d
d X d X d d d d d d d d d d d d X d d d ^ X X X X X     X X   X   X   v X X X X
X X X X X d X X d d X X X X d X X X d d ^ X d d X   X   X X X       X > v X X X
X X d d d d d X d d d X d d d d X d d d ^ < d X X   X X                 > > > V
```

## Acknowledgment

This project was supported in part by an Iowa State faculty startup grant and by the NSF SGER Grant IIS-0001542.