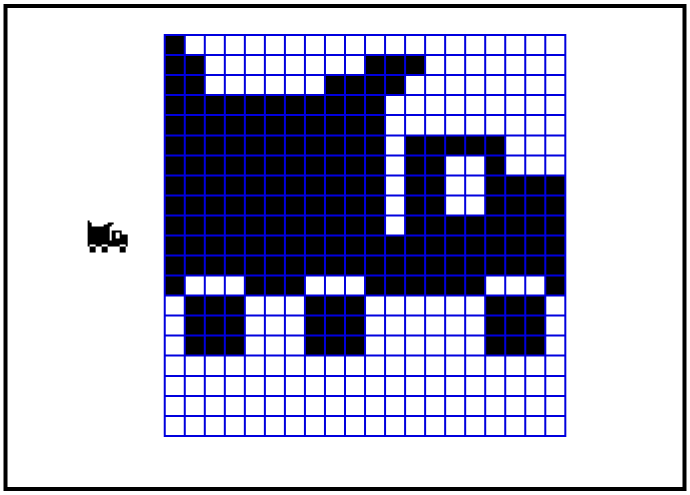
www.logofoundation.org

# A Full-Screen LogoWriter PrintShape Procedure

by

## Thomas F. Trocco

Computer & Science Department Chiar
Computer Teacher, Grades 2 - 8
St. Hilda's & St. Hugh's School
New York, NY



**Acknowledgements**

Thanks to Michael Tempel, Steve Neiman, and Becque Olson for their suggestions.

## Introduction

LogoWriter allows the user to change the turtle's shape to any of 30 or 90 additional shapes (depending on your platform). Some of these come ready-made with the program while others are blank. Any of these (except shape 0, the turtle) may be modified. Simply type **shapes** in your command center, and you'll be on your **shapes** page. By pressing the spacebar and using the arrow keys (or mouse) you can add or take away dots in a 16 X 16 grid (or 20 X 20 on the MAC). Users can then change the turtle's shape to any of these with the **setsh** command (e.g. **setsh 21**). Once a new shape has been chosen, it can be used just like the turtle (although you won't see it rotate as you change its heading). You can use these shapes for animation, for drawing, for **stamp**-ing and for **shade**-ing.

I introduce shape creation in the second grade, and have always run into two major problems which may sound familiar to you.

First, I introduce the **shapes** page to my students, they create a new shape, and they then want to print it out as they see it on the screen, in its full size. I cannot tell you how many times I've explained to students that the large size cannot be printed, but they still really desire to see a large version of their creation on paper. I've often thought it would be great to have a way to print out the page as it is seen on the screen.

Second, students want their shape to face the opposite way, but have trouble figuring out what dots to place. I've often thought is would be great to be able to have a print-out of the shapes page grid in both normal and reversed left/right for easier shape reversal. (LogoEnsemble and Macintosh LogoWriter users already have the ability to reverse their shapes on the **shapes** page).

And finally, for myself: While writing a paper on an ecology simulation, I realized that I wanted to have large grid printouts of my shapes for my readers so they could re-create my shapes if they desired.

So I sat down and tackled the task. The trick was to somehow read what pixels were used in a particular shape, store that information, draw a large grid on the screen, and fill in the appropriate spaces from the information stored. To read the information, I stamped the shape on the screen and tested for **colorunder**. To store the information, I created a variable list for each row of pixels, with '0' for off and '1' for on. What follows is an annotated program listing.

## PrintShape Procedure Program Listing

```
to ins.1
pr [PrintShape Procedure]
pr [(c) 1993 by Thomas Trocco]
pr [Computer Dept. Chair,]
pr [St. Hilda's & St. Hugh's School.]
pr [619 West 114th Street]
pr [New York, NY 10025]
pr "
pr [Written January 28-30, 1993]
pr "
pr [This program will display a shape in a large grid similar to
that seen on the flip side of a SHAPES page.]
pr "
end
```

This is the Master, or Super Procedure:

```
to printshape
clear.all
setup
ins.1
ins.2
set.array 0
grid
small
ins.3
check.set
check 0
make "row 0
set.fill
ins.4
ins.5
ins.6
end
```

This clears the graphics screen, command center, text screen, variable values, and hides the turtle:

```
to clear.all
rg
cc
ef
ht
clearnames
end

to ef
if not front? [flip]
ct
end
```

**Ct** (**cleartext**) is dangerous because if it is used in the command center while the flip side is showing, all text (i.e., all procedures) will be erased. I urge you to put this EraseFront procedure on each of your pages, and use **ef** instead of **ct**. This procedure will clear the text only after you have flipped to the front side.

If you are using Apple IIGS LogoWriter type this:

```
to setup
getshapes
make "lines 17
make "dist 160
end
```

If you are using LogoEnsemble type this:

```
to setup
make "lines 17
make "dist 160
end
```

If you are using MSDOS LogoWriter type this:

```
to setup
getshapes
make "lines 17
make "dist 160
end
```

If you are using Macintosh LogoWriter type this:

```
to setup
make "lines 21
make "dist 200
end
```

For Macintosh LogoWriter, also make these changes:

- in the procedure **small** change
  `setpos [-108 0]` **to** `setpos [-107 0]`
  and `label :shape` **to** `pd label :shape pu`
- in the procedure **grid** (occurs twice) change
  `setpos [-80 -80]` **to** `setpos [-80 -100]`
- in the procedure **set.fill** change
  `setpos list -73 (75 - :row * 10)`
  **to**
  `setpos list -63 (85 - :row * 10)`

**Setup** sets the size of the grid to be drawn on the screen (16 X 16 boxes, or 17 X 17 lines for MSDOS and Apple IIGS computers; 20 X 20 boxes, or 21 X 21 lines for the Macintosh) and the width of the grid (10 steps * the number of boxes).

```
to ins.2
pr [Please type in the number of the shape you wish to print.]
pr "
make "shape first readlist
ef
end
```

**Ins.2** waits for input and sets the variable **"shape** equal to it. **First** is necessary, even though only one number is being typed, so that **shape** is set equal to a number, rather than a one-member list. Without **first** you'll later receive a message such as

*setsh doesn't like [21] as input.*

```
to set.array :row
make word "pixels :row [ ]
if equal? :row :lines - 2 [stop]
set.array :row + 1
end
```

**Printshape** passes 0 as input to **set.array**. A variable (**pixels0**), made from the word **pixels** and the input, **0**, is given the value **[ ]** (an empty list). This procedure continues recursively, making more variables, **pixels1**, **pixels2**, etc. until **:line - 2** is reached (15 on MSDOS and AppleIIGS computers, 19 on Macintosh computers). Thus, one variable for each row is created. All have empty lists as their values.

```
to grid
pu
setc 2
setpos [-80 -80]
repeat :lines
[pd fd :dist bk :dist pu rt 90 fd 10 lt 90]
pu
setpos [-80 -80]
rt 90
repeat :lines
[pd fd :dist bk :dist pu lt 90 fd 10 rt 90]
end
```

**Grid** draws a grid in **color 2** on the screen with the dimensions **:lines** by **:lines** (17 X 17 on MSDOS and Apple IIGS computers , 21 X 21 on Macintosh computers.)

```
to small
pu
setc 1
seth 0
setpos [-140 0]
label :shape
setpos [-108 0]
pd setc 2
setsh :shape
stamp
setc 1
end
```

**Small label**s the number of the **shape** entered in **ins.2** and **stamp**s it in **color 2**; sets the **color** to **1** when finished.

```
to ins.3
type [Please wait while the shape is scanned....]
end
```

```
to check.set
pu
ht
setsh 0
seth 90
make "y 9
end
```

**Check.set** sets the **heading** of the turtle to 90deg., sets **:y** equal to 9 to be used in **check**.

```
to check :row
set.row :y
repeat :lines - 1
[ifelse equal? colorunder 2
[pd fd 0 pu make word "pixels :row se thing word "pixels :row 1]
[make word "pixels :row se thing word "pixels :row 0] fd 1]
make "y :y - 1
if not equal? :dist 200
[if equal? (int :y / 5) (:y / 5) [make "y :y - 1]]
if equal? :row :lines - 2 [stop]
check :row + 1
end
```

**Set.row :y** sets the position of the turtle as [-116 9], the upper left hand corner of the **stamp**ed **shape**. Next the turtle tests for **colorunder**. If the **colorunder** is 2 (i.e., if the **stamp**ed **shape** uses that position) a '1' is concatenated to the current value of the variable for that row (i.e., **pixels0**). If the **colorunder** is not 2 (i.e., if the **stamp**ed **shape** does not use that position), a '0' is concatenated to the current value of the variable for that row. The turtle then moves one step to the right and tests again. This continues for :**lines - 1** times (equal to the width of the shape in pixels). Next **:y** is decreased by 1. If the value of **:dist** is not equal to 200 (i.e., if the computer is NOT a Macintosh), **:y** is again decreased by 1 if the value of **:y** is a multiple of 5. This is necessary because on MSDOS and AppleIIGS computers, for every fifth vertical pixel that is addressed, the turtle doesn't move. This was designed into LogoWriter because pixels are vertical rectangles, not squares, and without this correction, shapes created with LogoWriter would be elongated vertically. Pixels are square on the Macintosh, so this correction is not applied to that computer. **Check** calls itself recursively so it can **check** the next row of pixels. **Check** will stop when the **:row** it is on is equal to **:lines - 2** (i.e., when it has checked all rows).

For example, **shape 0**, the turtle, has dots in the following pattern:

```
↓   R     O     W     S                 C     O     L     U     M     N     S     ->
            0     1     2     3     4     5     6     7     8     9     10    11    12    13    14    15


0
                                                      *     *
1
                                                *     *     *     *
2
                                                *     *     *     *
3
                                                      *     *
4
                        *     *                       *     *                       *     *
5
                              *     *     *     *     *     *     *     *     *     *
6
                                    *     *     *     *     *     *     *     *
7
                              *     *     *     *     *     *     *     *     *     *
8
                              *     *     *     *     *     *     *     *     *     *
9
10                            *     *     *     *     *     *     *     *     *     *

11                                  *     *     *     *     *     *     *     *

12                            *           *     *     *     *     *     *           *

13                      *     *                       *     *                       *     *

14

15
```

As the turtle moves from left to right, scanning each position in the 16 X 16 grid, the following values would be stored:

↓ ROWS COLUMNS ->

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Variable Name |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Pixels0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Pixels1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Pixels2 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Pixels3 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Pixels4 |
| 5 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | Pixels5 |
| 6 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | Pixels6 |
| 7 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Pixels7 |
| 8 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | Pixels8 |
| 9 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | Pixels9 |
| 10 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | Pixels10 |
| 11 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Pixels11 |
| 12 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | Pixels12 |
| 13 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Pixels13 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Pixels14 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Pixels15 |

These variables will later be used to fill in the squares on the grid.

```
to set.row :y
setpos list -116 :y
end

to set.fill
pu
setpos list -73 (75 - :row * 10)
end
```

**Set.fill** places the turtle in the proper position (upper left corner) to begin filling in squares on the grid.

```
to ins.4
cc
type [|Do you want your shape (N)ormal or (R)eversed left-right|]
type char 13
ifelse equal? readchar "n [cc fill.normal] [cc fill.reverse]
end
```

**Fill.normal** will read the variable lists from left to right; **fill.reverse** will read them from right to left.

```
to fill.normal
if equal? first (thing word "pixels :row) 1
[pd fill pu]
fd 10
if (count thing word "pixels :row) > 1 [make word "pixels :row bf
thing word "pixels :row fill.normal stop]
if :row < :lines - 2 [make "row :row + 1 set.fill fill.normal]
end
```

If the **first** member of the list is a '1', the square will be filled. The turtle moves 10 steps to the right. If the **count** of the number of items in the list is greater than 1, the **first** member of the list is removed (with **ButFirst**), and the new shorter list is passed to **fill.normal**. This continues across the row until the last item is used. **:row** is increased by one; the turtle is moved down one row (**set.fill**), and the procedure is called recursively, filling in the next row. This continues until all rows have been filled (when **:row = :lines - 2**)

```
to fill.reverse
if equal? last (thing word "pixels :row) 1
[pd fill pu]
fd 10
if (count thing word "pixels :row) > 1 [make word "pixels :row bl
thing word "pixels :row fill.reverse stop]
if :row < :lines - 2 [make "row :row + 1 set.fill fill.reverse]
end
```

**Fill.reverse** is identical to **fill.normal**, except that the list is read backwards (with **last**), and the **last** member is removed each time (with **ButLast**).

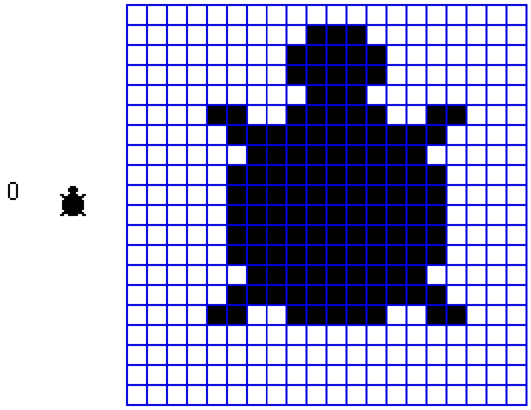
```
to ins.5
cc
type [Type "P" to print this screen.]
if equal? readchar "P [printscreen printtext]
cc
end
```

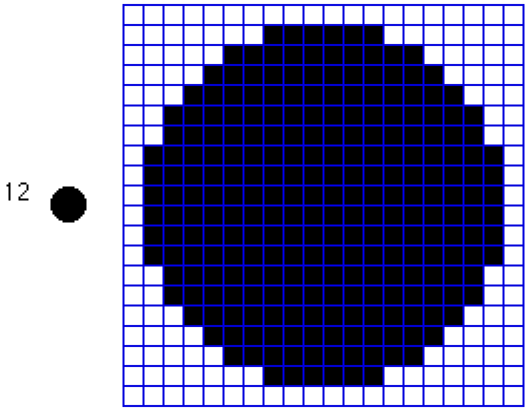
If a "P" is typed, the screen will be printed. The **printtext** command is there to force a form feed for those printers that need it.

```
to ins.6
cc
type [Type "Y" if you would like to see another shape.]
ifelse equal? readchar "Y [printshape] [cc]
end
```

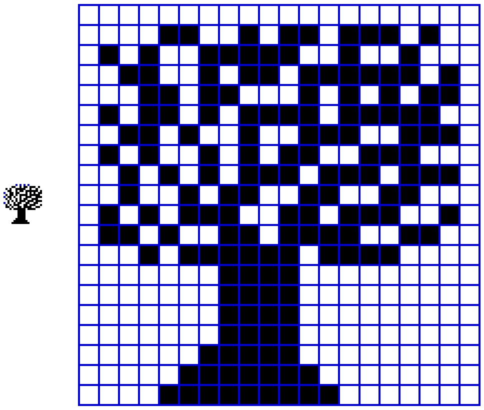
If a "Y" is typed, **printshape** will be **run**. Otherwise the command center is cleared and the program ends.

Following are some sample **shape** printouts, normal and reversed.

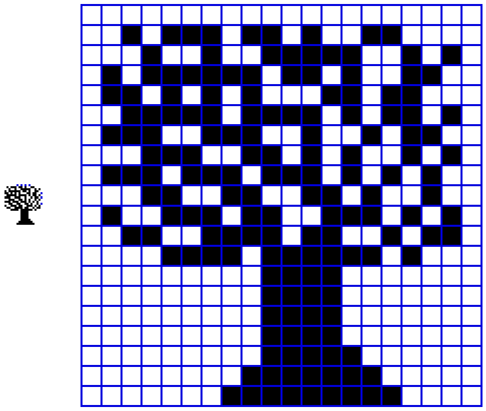




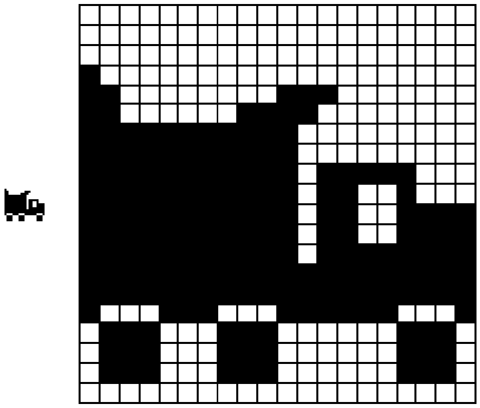
23



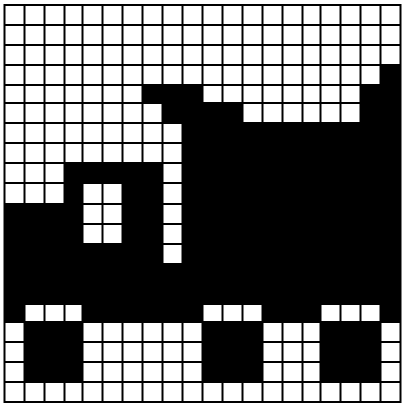
23



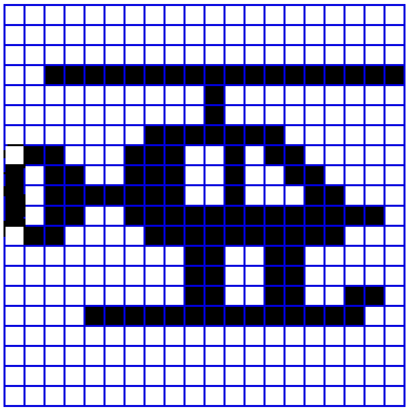
26



26



25



25