BASIC for Blind Users

By Ray McAllister

Introduction

More and more blind or visually impaired people are using computers and entering the work force. Programs that are not accessible to the blind will not benefit this part of the population. In many situations in the United States, the law requires that work places accommodate the special needs of the disabled. This includes computer software. It is, therefore, necessary for software to be written in a way that the blind can use it. As a blind person, myself, and one experienced with using BASIC, I am offering this article to show programmers some simple and doable ways to make their software accessible to the blind.

I can say with confidence that these techniques do work. I've tested programs employing them on WindowEyes, my screen reader, and I've had other blind people test them with other screen readers, including JAWS, one of the most commonly-used screen readers. In all cases, the BASIC programs are very accessible.

In this article, Just BASIC is referred to as JB, Liberty BASIC, LB
Finally, before getting going, it must be noted that the term "blind" refers to many different types of conditions. I am totally blind, which means I have no light perception. Actually, I have no eyes. They are both prosthetic. My computer needs, then, are different from someone who has limited vision. Nonetheless, writing software that is accessible for the totally blind will cover all types of blindness.

How Screen Readers Work

A blind person may use text-to-speech technology to have the computer read the screen. One may also use a refreshable Braille display, which turns the text on the screen into raised dots of magnetic pins or the sort. However the information is handled, the principles are the same. Screen readers read text, not graphics or designs.

Another aspect of screen readers has to do with the mouse. Since the mouse moves around here and there with little orientation outside of the screen, screen readers replace mouse controls with keyboard commands. One uses complex sets of hot keys to move the mouse pointer around. Often, the mouse pointer, then, can be used to locate things on the screen to read.

This information can lead one to some very logical conclusions. First, complex graphics and animations are not recognizable to those totally blind. Obviously, video games, then, would be impossible to make accessible at this time in earth’s history unless a blind character was designed who would have special sonar powers. Nonetheless, my focus here is on making programs accessible that a blind person might

actually use, like one for data entry at a job.

Next, consider how keying around the screen with the mouse pointer to look for useful text can take time. As a result, blind people really like keyboard shortcuts and hot keys. A program can still have mouse-driven functions for sighted users. There should also be keyboard short-cuts one can learn to quickly get things done.

Programming with the Blind in Mind

It is now time to consider how to write programs that the blind can use easily. The main principle to keep in mind is simplicity. If you can write a program that mainly just uses the main window, that's perfect. That window is easily searchable by using just the arrow keys or ctrl-f for string searches. Data can be entered with the "input" or "input$" commands, and screen readers handle that excellently. It is also easy to use the input$(1) system so one simply presses certain keys for things to happen. If a program is only going to be used by the blind, this may be enough for good programming. Otherwise text windows and text editor windows are about as accessible, if not, more, depending on what you are doing.

It can also be helpful to have sound effects now and then to tell a blind person what is happening. I wrote a program designed to help the blind search the Greek and Hebrew Biblical texts easily, using JB. I recorded a Wave file of my turning pages in a book which would be heard when the program is searching the Bible files for hits. When all the hits are found and displayed, a trumpet fanfare plays. The sound effects made it so I wouldn't have to sit in front of the computer browsing the screen while waiting until a search was done. I could just sit back and relax, following the sounds. With LB's ability to handle DLL calls, MIDI beeps and sounds could be included in a program and be more space efficient than Wave files in certain situations. A program could be written so that sound effects could be turned on or off if desired.

In addition, there is a speech dll for LB that will convert any string to speech. You can find it at http://basic.wikispaces.com/SpeechLibrary and it works well. You can add a speech option to programs you write.

The controls for windows like comboboxes, listboxes, textboxes, radiobuttons, etc., are easily recognizable by screen readers. As the arrow keys move the cursor through a list, the screen reader reads the items. Even text entered into a graphics window via the "\" command is recognized well, but the blind person must key the mouse pointer to the message. Text that is converted to bitmap files or sprites is not recognizable by screen readers and should only be used for programs not intended for use by the blind.

Where accessibility becomes a problem is when the focus must be set on a graphics window for mouse and/or Inkey$ work. The Inkey$ option is nice for the blind because it does allow one to control the computer through key strokes. If, however, there is a combobox on the screen that a blind person needs to access, the focus often returns right back to the graphics area before the key-driven mouse pointer can do very much. It begins to feel as if the computer has Attention deficit Disorder. This is where a little creative coding must be done, but it is not difficult. I even retrofitted LB's Piano 6 to get around this problem, and

I’m not even the one who originally wrote the program.

The program must be written so the focus can be changed to either the combobox or the graphics window. In the place in the program where all the Inkey$ work happens, a hot key must be provided to transfer focus out of the graphics window. An easy way to do this is with a variable you might call, fc, for focus. When fc = 0, the focus is on the graphics window. When fc = 1, the focus is on the combobox. First, then, it must be established at the very beginning of the program, before any windows open, that fc =0. Then, a command for Inkey$ might read:

```
If Inkey$ = “ “ then fc = 1
```

This sets the space bar as the hot key to remove the focus from the graphicbox. Any place where the focus is to be set on the graphics window, then, is coded this way, adjusting only for whatever might be happening in the graphics window:

```
If fc = 0 then
#window.graphicbox “setfocus”
#window.graphicbox "when leftButtonDown [branchLabelA]"
#window.graphicbox "when leftButtonUp [branchLabelB]"
#window.graphicbox "when leftButtonMove [branchLabelC]"
#window.graphicbox "when characterInput [branchLabelD]"
Else
#window.combobox “setfocus”
End if
```

Of course, if no mouse activity is needed, the button moving commands wouldn't be there. When fc = 0, then the focus is on the graphicbox. When fc = 1, the focus is on the combobox. Since the only options for fc are 0 and 1, this coding structure is fine.

Finally, there must be a way to return the focus to the graphicbox. To do this, one can place a button near the combobox with a message on it like “select” or “choose.” A blind person can use tab or shift-tab to move over to that button. Clicking that button would take the program to a branch label that, among anything else that must be done, sets fc to equal 0.

You can actually test this keyboard access, yourself, with your own keyboard. I tried my modified Piano6 without my screen reader running, and, while I didn’t have a clue which instrument I was choosing, I was able to freely move about the list and choose whatever item.

I must also mention a most obvious way to make things accessible. The "run" command lets one run other programs in the computer. One could have some information saved as a text document and then have BASIC open that file in NotePad, which is highly accessible. It's a good last resort option.

Conclusion

The information here is not intended to solve every possible accessibility problem. The solutions offered may not work in every case. You may think of other ideas that work better. What I want most is for the principles behind these ideas to be in your mind as you write programs. The blind can greatly benefit from well-written software. JB, LB, and RB can be used to write such well-written software. It is my challenge, then, that if a program you write might possibly be used by a blind person, the program should be written to be accessible for such a person.

Appendix A.

Sample code, good in JB or LB.

Below I present a sample program I wrote that is a blind-accessible graphics analyzer. Yes, it can be done. Blind programmers will want to pay close attention, here, as well.

Screen readers can handle the main window writing best, and the "\" messages, statictext box, combobox, and button, easily. The line saying that one can press the space bar for accessible combobox displays in white on white, invisible to the eye, but visible to screen readers. this reduces clutter, some.

This program helps one explore the "rule" comand in graphics work. Normally, when someone puts, for example, a blue pixel where a red pixel was, the blue writes over the red, and all one sees is blue. That's the default setting, known as "rule over." There are many other things that can happen, like colors mixing in rather interesting ways.

This program works by having self-described writing on backgrounds on top the screen. The writing tells what color it is and what color the backgrounds are. The main background on the top of the screen is black. Then, that area is covered by a filled box, default color, hot white. Because the xor rule is used, as the default, there appears to be no change in the color behind, what I refer to as the "tinted glass." That all will change with the touch of any of several buttons.

This program explores the 16 rules. A combobox lets one choose which rule to use, and hot keys let one change the color of the "tinted glass" that covers the writing. When one presses "space," whether sighted or blind, the focus locks on the combobox until the "select" button is pressed. this allows one to use the arrow keys to freely move around the combobox. After "select" is pressed, focus returns to the graphicbox for Inkey$ work to pick up keyboard commands. I would have no objection to someone's editing this program for mouse control. As long as the existing commands aren't disturbed, it won't be any trouble for the blind.

A status line at the bottom of the graphicbox shows which rule is being used, the color of the "tinted glass," and which color(s) one is changing with numbers or arrow keys. (Some color-change options, like "cyan" or "all colors" allow for more than one color to be controlled.) When "p" is pressed, a print-out of current

RGB values appears in the main window. Actually, when the program starts up, a print-out of that type also appears, but with the actual colors of the writing. Now, while the same color names are shown, whatever the new RGB value is of that color appears afterward. So, while the first print-out says "Red: 255 0 0," it may read something like this, later: "Red: 0 255 255." When "p" is pressed, a ding is sounded by calling up a nonexistent Wave file. This is because no notification of this print-out happens on the status line.

Someone may have better ideas for screen layout, as far as eye-appeal goes. Moving things around for eye-appeal shouldn't mess anything up. One could even set the screen up so the list of hot-keys only appears when the user presses a "help" key like "ctrl-h." That would remove clutter the rest of the time. Whatever is done, just press "space" and see if it still works right.

Finally, when one chooses "CopyPen (Over)," that is the same as the standard, default "Over" setting where the new pixel color writes over the old. When this setting is used, the top area, where the colored writing is usually seen, becomes a color test lab. Between the arrow keys, moving one notch at a time, and the number keys, jumping to spots from 0-255, one can quickly key in any RGB value. This is helpful for one trying to find the best RGB set for a color. A blind person can very easily have a sighted friend look on and say when the color is best. Actually, the blind programmer could just have the sighted friend sit at the keyboard and key in the right color. I've had my wife help me with this when I wanted just the perfect color. I can, then, read the RGB values on the status line.

So, let's explore the world of "rules" in BASIC. Appendix B contains a brief piece I found online explaining the rules and what they do.

Code

```
'BASIC Rules
'A blind-accessible graphics analyzer.
'Colored writing is covered by a box (tinted glass) with various color
s and rules invoked.
'Notice all the keyboard commands.
'The main window, "\" messages, statictext, and combobox are all scree
n-reader friendly.
'The blind user uses the space bar to lock focus on combobox for rule
change.
'The announcement about this is white on white, only visible to screen
 readers.
'Main window print-
out lists colors as drawn and RGB values after rule change.
'A ding confirms that RGB print-out happened in main window.
'Status line at bottom of graphicbox tells what's presently happening.
'Rule "CopyPen (Over)" turns the colored box into an rGB color testing
 lab.
```

```
cls
[SetVariables]
test = 0
dim rule$(16)
dim rnum(16)
dim name$(7)
dim color(3)
fc = 0  'Focus control variable
rule = 1
choice = 1
name$(1) = "red."
name$(2) = "green."
name$(3) = "blue."
name$(4) = "cyan."
name$(5) = "yellow."
name$(6) = "magenta."
name$(7) = "all colors."
color(1) = 255
color(2) = 255
color(3) = 255
rgb$ = "100"
[readRuleNameData]
for k = 1 to 16
read a, a$
rnum(k) = a
rule$(k) = a$
next k
'Window setup
WindowWidth = DisplayWidth
WindowHeight = DisplayHeight
   statictext #g.s, "Choose Rule", 10, 505, 300, 100
combobox #g.c, rule$(),[select], 400, 505, 300, 200
graphicbox #g.g, 0, 0, 1333, 500
button #g.b, "Select", [choice], ul,900, 550, 100, 80
open "BASIC Rules" for window as #g
#g.s "!font 20"
#g.c "font 20"
#g.b "!font 20"

#g.c "select ";rule$(1)

#g "trapclose [quit]"
[work]
'Graphicbox message setup
#g.g
"place 0 0 ; Color Black ; BackColor Black ; down ;BoxFilled 1332 200"
```

```
#g.g "font 40 70 ; down ; place 0 70 ; Color Red ; BackColor Green"
#g.g "\Red on green"
#g.g
"place 0 140 ; down ; Color 255 160 0 ; BackColor Blue ; font 40 70"
#g.g "\Orange on blue"
#g.g
"place 680 70 ; down ; Color Yellow ; BackColor 140 0 200 ; font 40 70
"
#g.g "\Yellow on purple"
#g.g
"place 680 140 ; down ; Color 100 100 100 ; BackColor White ; font 40
70"
#g.g "\Grey on white"
#g.g "place 0 250 ; down ; Color Black ; BackColor White ; font 20"
#g.g "\Press A for control of all colors of the tinted glass."
#g.g
"\   R for red; G, green; B, blue; C, cyan; Y, yellow; M, magenta."
#g.g
"\   0-9 to change selected color(s) to 1 of 10 values from 0-255."
#g.g "\   Arrow Keys to advance color(s) 1 notch at a time."
#g.g "\   P to print to screen current RGB list."
#g.g "\   Alt-F4 to close window."
'Make accessibility line visible only to screen readers.
#g.g "color white ; backcolor white ; down"
#g.g "\   Space Bar for blind-accessible combobox."
#g.g "flush messages"
#g.g "\"
#g.g "flush change"
gosub [printOut]
gosub [ruleChange]
[keyCheck] 'Get computer watching for key-presses.
scan
'Blind-accessibility focus control.
      if fc = 0 then
            #g.g "setfocus"
            #g.g "when characterInput [keyWork]"
else
#g.c "setfocus"
end if
wait
[keyWork] 'Act according to key presses.
k$ = Inkey$
key = asc(k$)
select case
case k$ = " "
fc = 1  'Allow blind users to have focus locked on combobox.
```

```
case (key >= 48) and (key <= 57) 'Number key control.
'Change only the selected values.
for k = 1 to 3
if mid$(rgb$,k,1) = "1" then color(k) = int(val(k$)*(255/9))
next k
gosub [ruleChange]
'Arrow key control.
case (asc(right$(k$,1)) =_VK_LEFT) or (asc(right$(k$,1)) =_VK_DOWN)
show = 0 'Status line and screen don't update if no change.
for k = 1 to 3
if mid$(rgb$,k,1) = "1" then color(k) = color(k) - 1
if color(k) < 0 then
color(k) = 0
else
show = 1
end if
next k
if show = 1 then gosub [ruleChange]
case (asc(right$(k$,1)) =_VK_RIGHT) or (asc(right$(k$,1)) =_VK_UP)
show = 0
for k = 1 to 3
if mid$(rgb$,k,1) = "1" then color(k) = color(k) + 1
if color(k) > 255 then
color(k) = 255
else
show = 1
end if
next k
if show = 1 then gosub [ruleChange]
'Set which RGB value(s) of tinted glass cover to change.
case upper$(k$) = "R"
rgb$ = "100"
choice = 1
gosub [ruleChange]
case upper$(k$) = "G"
rgb$ = "010"
choice = 2
gosub [ruleChange]
case upper$(k$) = "B"
rgb$ = "001"
choice = 3
gosub [ruleChange]
case upper$(k$) = "C"
rgb$ = "011"
choice = 4
gosub [ruleChange]
```

```
case upper$(k$) = "Y"
rgb$ = "110"
choice = 5
gosub [ruleChange]
case upper$(k$) = "M"
rgb$ = "101"
choice = 6
gosub [ruleChange]
case upper$(k$) = "A"
rgb$ = "111"
choice = 7
gosub [ruleChange]
'Send RGB print-out to main window with confirming ding.
case upper$(k$) = "P"
playwave "nofile.wav", async
gosub [printOut]
end select
goto [keyCheck]
[select]  'Access data from combobox.
#g.c "selectionindex? rule"
gosub [ruleChange]
'Blind accessibility focus control.
      if fc = 0 then
            #g.g "setfocus"
            #g.g "when CharacterInput [keyWork]"
else
#g.c "setfocus"
end if
wait
'Button action
[choice]
fc = 0
goto [select]
wait
[quit]
close #g
print
print "Press Alt-F4 to end program."
wait
[ruleChange]
'Remove what graphics will be changed.
#g.g "delsegment change ; redraw"
'Determine which rule to use.
#g.g "rule ";rnum(rule)
'Change message colors as directed.
#g.g "Color ";color(1);" ";color(2);" ";color(3)
```

```
#g.g "BackColor ";color(1);" ";color(2);" ";color(3)
#g.g "place 0 0 ; size 1 ; down ; BoxFilled 1332 200"
'Revise status line at bottom of graphicbox.
message$ = "Rule = ";rule$(rule) ; ", color ";color(1); " "
;color(2);" ";color(3);", and changing ";name$(choice)
#g.g "font 20 ; place 0 485 ; Color 0 0 128 ; BackColor White ; down"
#g.g "\" ; message$
                                              'Flush new material.
#g.g "flush change"
'Blind accessible focus control.
      if fc = 0 then
            #g.g "setfocus"
            #g.g "when characterInput [keyWork]"
else
#g.c "setfocus"
end if
return
'Set in main window RGB values of messages through tinted glass.
[printOut]
'Notify differently for first print-out.
if test = 0 then
print "Original print-out, actual color values."
else
print
print "Rule = ";rule$(rule) ; ", glass color ";color(1); " "
;color(2);" ";color(3)
end if
test = 1
'Use col$ function to get pixel values.
print "   Red: ";col$(7,40)
print "   Orange: ";col$(5,110)
print "   Yellow: ";col$(705,40)
print "   Green: "; col$(6,40)
print "   Blue: ";col$(4,110)
print "   Purple: ";col$(706,40)
print "   White: ";col$(755,110)
print "   Grey: ";col$(756,110)
print "   Black: ";col$(622,110)

        'Blind accessibility focus control.
                     if fc = 0 then
            #g.g "setfocus"
            #g.g "when characterInput [keyWork]"
else
#g.c "setfocus"
end if
```

```
return
[listOfRulesWithCodeNumbers]
data 10, "NotXOrPen, (XOr)", 7, "XOrPen", 9, "MaskPen", 8,
"NotMaskPen"
data 3, "MaskNotPen", 5, "MaskPenNot", 15, "MergePen", 2,
"NotMergePen"
data 12, "MergeNotPen", 14, "MergePenNot", 6, "Not", 13,
"CopyPen (Over)",
data 4, "NotCopyPen", 1, "Black", 16, "White", 11, "NoOperation"
'Function col$ -- col$(x,y) finds RGB value for pixel x,y.
function col$(colorx,colory)
#g.g "getbmp gpv ";colorx;" ";colory;" 1 1"
bmpsave "gpv", "getpvaluetemp.bmp"
open "getpvaluetemp.bmp" for input as #gpv
colorresult$ = input$(#gpv, lof(#gpv))
close #gpv
if asc(mid$(colorresult$, 29, 1)) = 32 then
red = asc(mid$(colorresult$, 69, 1))
green = asc(mid$(colorresult$, 68, 1))
blue = asc(mid$(colorresult$, 67, 1))
end if
col$ = str$(red) + " " + str$(green) + " " + str$(blue)
end function
```

Appendix B.

Note: Before reading the article below, I must say this: The article says that XOrPen is the same as Rule XOr in JB or LB. After running many tests with BASIC Rules and other programs I've written, it seems that NotXOrPen is actually the same as Rule XOr. In addition, the rule named in the program in Appendix A as "NoOperation" is listed as "NOP" below.

Article follows:

BINARY RASTER OPERATIONS

Taken from http://www.libertybasicuniversity.com/lbnews/nl101/4.htm

The Liberty BASIC drawing rules are examples of Binary Raster Operations, sometimes referred to as ROP2. Each raster-operation code represents a Boolean operation in which the values of the pixels in the selected pen and the destination image are combined. The manner in which they are combined is described below. The destination image in this case means the graphics that are contained in a graphicbox or graphics window at the time the drawing is accomplished.

Here is the Liberty BASIC syntax for setting the drawingrule:

print #handle, "rule rulename"
print #handle, "rule xor" 'as it would appear in a program
print #handle, "rule over" 'as it would appear in a program
or
print #handle, "rule "; _R2_NOTXORPEN 'as it would appear in a program

This command specifies whether drawing overwrites (rulename OVER) on the screen or uses the exclusive-OR technique (rulename XOR). You can also use Windows constants to select a drawing rule (as shown above). Note that the Liberty BASIC named rules, OVER and XOR must be placed within the quote marks. If Windows constants are used for the drawing rule command, they must be preceded by an underscore and placed outside of the quote marks.

The most-used rules are the two named Liberty BASIC rules: OVER and XOR. OVER causes pixels drawn with graphics drawing commands such as LINE, BOX or CIRCLE to appear in the currently selected COLOR, replacing the pixels on the screen. XOR combines the color of the drawing pen with the pixels on the screen using the exclusive-OR operation. Read below for more about using RULE XOR. The drawing rule can be changed at any time during graphics drawing, and affects only the graphics drawn after the rule is changed. The rule can be changed many times, creating an unlimited number of interesting or unusual effects. Look below for an explanation of the binary raster operations, then experiment with them in your graphics to see the possibilities.

Here are the constants that Windows defines:

_R2_BLACK
Pixel is always 0.

_R2_COPYPEN Liberty BASIC's rule OVER.
Pixel is the pen color.

_R2_MASKNOTPEN
Pixel is a combination of the colors common to both the screen and the inverse of the pen.

_R2_MASKPEN
Pixel is a combination of the colors common to both the pen and the screen.

_R2_MASKPENNOT
Pixel is a combination of the colors common to both the pen and the inverse of the screen.

_R2_MERGENOTPEN
Pixel is a combination of the screen color and the inverse of the pen color.

_R2_MERGEPEN
Pixel is a combination of the pen color and the screen color.

_R2_MERGEPENNOT
Pixel is a combination of the pen color and the inverse of the screen color.

_R2_NOP
Pixel remains unchanged.

_R2_NOT
Pixel is the inverse of the screen color.

_R2_NOTCOPYPEN
Pixel is the inverse of the pen color.

_R2_NOTMASKPEN
Pixel is the inverse of the R2_MASKPEN color.

_R2_NOTMERGEPEN
Pixel is the inverse of the R2_MERGEPEN color.

_R2_NOTXORPEN
Pixel is the inverse of the R2_XORPEN color.

_R2_WHITE
Pixel is always 1.

_R2_XORPEN Liberty BASIC's rule XOR Pixel is a combination of the colors in the pen and in the screen, but not in both.

RULE XOR
Rule XOR gives us a very handy ability. It allows us to draw graphics, then to draw them again later in the same spot and erase them, leaving the pixels on the screen as they were before the graphics were drawn. Look in the FreeForm program code that comes with Liberty BASIC. Objects are drawn on the screen and moved about using rule XOR. While an object is being moved by the user, it is drawn once to be displayed on the screen. When the mouse moves as the user drags the object, it is drawn a second time at the same location to "erase" it and restore the screen to its previous appearance. It is then drawn in its new location, only to be drawn again on that spot to "erase" it when it is moved again. This continues as long as the object is being moved by the mouse. When it is in the desired position, the rule is changed back to rule OVER and the object is drawn again, not to be erased this time.

Freeform is a large and complex program. For a simple example of rule XOR at work, run the following small demo program. It begins with some text on the screen. When the "Do Change" menu is clicked, graphics are drawn using rule XOR.

Click the "Do Change" menu again, and the screen is restored to the simple text it displayed at the start. The graphics have been "erased", leaving the appearance of the screen exactly the same as it was before the drawing took place. Notice that when using rule XOR, drawn colors are dependent upon the existing colors on the screen. "Red" will not necessarily appear red. A "red" line may change colors many times as

it passes over pixels of different colors that are already on the screen. Change the colors and backcolors in the code below to see how color is affected by rule XOR. Rule XOR is not the best choice when consistency of colors is needed. It is the best choice when temporary graphics must be drawn and removed, leaving the screen appearance unchanged.

```
nomainwin
UpperLeftX=100:UpperLeftY=10
WindowWidth=400:WindowHeight=440
menu #1, "&Change","&Do Change",[doChange],_
    "E&xit",[quit]
open "XOR Demo" for graphics_nsb_nf as #1
print #1, "trapclose [quit]"

'drawn with default OVER:
print #1, "down; color blue; backcolor yellow"
print #1, "fill yellow; size 5"
print #1, "place 20 140"
print #1, "font arial 20"
print #1, "\XOR Drawing Rule Demo!"

'switch to XOR:
print #1, "rule xor"
wait

[doChange]
print #1, "place 240 240"
print #1, "color red; backcolor cyan"
print #1, "circlefilled 140"

print #1, "place 10 70"
print #1, "color darkgreen; backcolor white"
print #1, "boxfilled 210 240"

print #1, "place 210 100"
print #1, "color blue; backcolor pink"
print #1, "ellipsefilled 100 125"

wait

[quit]
close #1:end
```

Conclusion | Appendix A. | Code | Appendix B.