Basic Programming for Blind Programmers

By Ray McAllister

As a blind person, I have found many tricks that help me code with the BASIC platforms, Just BASIC, Liberty BASIC, and Run BASIC. In this article, I discuss how one with no sight can write useful programs with these languages, even how one, totally unsighted, can code graphics and animations. In this article, Just BASIC is referred to as JB, Liberty BASIC, LB, and Run BASIC, RB.

I also must make it clear, here, that if you have no experience coding in BASIC, some of this information may be over your head. In that case, read these paragraphs, not to understand everything, but to get a vision of the possibilities of what you can do. As you learn more about BASIC, refer back to this article for further insight.

General Information

Before discussing graphics, I should share some general information and insights. First, JB has not been updated in several years. This means that users of later versions of Windows like Vista and Windows 7 may face a few minor challenges that are easy to handle. For such people, installing JB in the default place in Programs & Features, means you'll have to run it as administrator all the time to do anything. Because of this, I have my JB installed in my C-drive main folder on my hard drive as some other people have done and recommended. LB and RB don't have this problem as they are up to date with Windows. Just have them installed where the installation guides recommend.

Next, it seems that the help files for LB and RB are much more accessible than those in JB. This is probably because versions of Windows starting with Vista and Windows 7 don't come equipped to handle those types of help files. This is no problem, though, because a quick google search for a command or list of commands in JB yields more than enough information. Just do a google search for what you're looking for and "Just BASIC," and you're well on your way. The tutorial files that come with JB, LB, and RB, are all easily accessible as text documents in the file folders and are all very helpful. JB users might also try downloading LB as a trial version. The help files are very accessible there, and, even after the trial period expires, they will still open if reached via the file folders without using the LB program to reach them. Some things, though, like API/DLL work, the "sort" command, and "stringwidth?" won't work in JB.

In addition, there are utilities one can download from Microsoft to enable one to use these kinds of help files in later versions of Windows. I found mine at http://www.askvg.com/how-to-open-help-files-hlp-in-windows-7-that-require-windows-help-program-winhlp32-exe/ and it helped. That page has files for Windows 8 on it also. Once I installed the right file, I could browse the help/tutorial menus in JB to find the files, on various topics, but the files, themselves, still wouldn't read with my screen reader. I found, though, that I could use ctrl-a and ctrl-c to copy the text to my clip board and read it fine in MS Word. These web sites give you two options for files to download, one for 64-bit Windows and one for 32-bit

Windows. If you check in your C-Drive, for the "programs" folders, and there is a "Programs" folder with an "86" at the end of the name, the 64-bit download should work. If that folder in your C-drive has some other number, try the 32-bit download first. If it's the wrong download, your computer just won't accept it, and you must try the other one.

The code editors are excellent with screen readers. LB's hot-key of alt-G for a quick jump to any part of the program is especially handy. Control-F, in any of these platforms, will enable you to search for a word or phrase in the program, too, just as it does about anywhere else in Windows. In addition, a browse through my article on writing programs to be used by the blind will enable you to write good programs that you, yourself, can use.

I have not found the debugger to be very helpful, though, with screen-readers. This is not a huge problem, though. What the debugger does is let one follow the value of a variable like X through a program. I coded in BASIC without a debugger just fine for years before these platforms. It is fairly simple to put a "Print X" statement in strategic places in a program and use the very accessible main window editor to review the results or save them to a file if you wish.

A blind person can also fairly easily reverse-engineer the coding in Piano-6, in LB, to learn how to code MIDI programs to make all kinds of fun music. I did one where hearts are dancing in time to music I coded that way.

The quick application generator in LB I did not find very efficient for using with my screen reader. I found that using the tried and true method of making a stand-alone program in JB works fine. I have stand-alone template folders in my computer, one for JB, one for LB. They have all the required files in them. When I want to make a program that will run on anyone's computer, I create a new folder, first, and then copy the contents of the right template folder into the new folder. then, I make a tkn form of the code file and put it in the new folder. Then I just change the ...run.exe file to the same name as the tkn and add any other files like wav or bmp files that I need to add. It's actually quite fast.

Graphics Coding

I devote the next major section of this article to making windows and graphics. First, I'll say simply and plainly that you will want to pay very close attention to the XY coordinates you enter. (X is the horizontal, and Y is the vertical.) It is also helpful to have a sighted friend "proof-read" the screen when a program runs, though you will find less and less need for that as you get more experienced. I also must add–-and I don't believe this is JB, LB, or RB's fault, but the colors you can code for by name don't always look like their names. Blue, black, white, and green are fine. The two that sighted friends tell me don't look as accurate are brown and pink. Brown looks more greenish, as it is an even mix of dark green and dark red. Pink is more purplish as it is an even mix of blue and red. I'll just tell you here that for these two colors, use a graphics window and use the RGB value option to choose the red, green, and blue values precisely. A good tree-trunk brown is "83, 53, 10" and a good pink is "255, 128, 128" It is easy to google for RGB values of colors you want to code correctly. A sighted friend can also help you tweak the values a little.

As you may be noticing, RGB colors don't work like paint or crayon colors. They work by the rules of mixing light. A value of 255 is the brightest, and 0, no expression of color. Red 255 and green 255 make yellow. To make orange, turn the green down, some. All three colors on full make white. Nothing is black. So, to turn red to pink, keep the red high while adding blue and green in equal values. Looking through a list of RGB color values and experimenting a little may make this clearer.

Now, let's look at how to draw pictures, blind. For coding graphics, you will probably wish to use the graphics template I am including as Appendix A. For that, you won't have to keep writing the same things over and over and over. You just start your coding at the section called [work]. Whatever you do, I recommend using DisplayWidth and DisplayHeight for the dimensions of graphics windows, at least, in the initial stages of work. This way you have a large screen 1333 pixels across and 698 pixels down to work with. Next, look through all the commands for graphics, noting ones most helpful for a blind programmer. Then, just have fun. You'll be reading a lot in the next paragraphs about how a knowledge of mathematics is helpful when coding graphics. While this is true, you can code a simple stickman simply using dots, lines, circles, and boxes, all codable with quick and easy commands and no long formulas.

It is important to know how the graphics you code are laid out. The default setting is that anything new you code writes over anything already there. Thus, if you code a red rectangle and then a blue circle that overlaps part of the red rectangle, the red in the rectangle will be written over with blue in the places where the blue overlaps. It is just as if someone set a blue cut-out of a circle over a red cut-out of a rectangle. This can be helpful as you don't have to worry about colors mixing as you would if you were using paint or crayon. It also affects the order in which you draw things. If you are coding an angel, frontal view, with wings coming out from behind, you will want to draw the wings first and then the body over them. If, however, you are doing a back view, you will code the body first and the wings last, as they are showing up front, now. If you explore the "rule" command, you can change how this all works and get colors to mix and reorganize in various ways.

There are many user functions, some I found and some I wrote, that are very useful for a blind programmer. The first one, col$, finds the RGB values of any pixel on the screen. Because this does so by using bitmaps, it will run on JB as well as LB. The command ' print col$(100,200) ' returns the RGB values as a string of characters for the pixel at coordinates 100/200. The DLL option for pixel color checking in LB can be helpful also and run faster. Just see which works best in the situations you encounter.

With col$ you can code some graphics and then write a for-next loop to return the values of all the pixels from 100/100 to 600/100, and have it print the needed X value followed by the color of that pixel. That is a great way to check your work without help from a sighted person. Appendix C contains code for a program that will draw a heart. The explanation guides you through using col$ and other useful tricks for coding graphics.

While simple pictures can be drawn with lines, boxes, circles, etc., a blind programmer must rely a lot on mathematics to code more complex graphics. Sine, cosine, and tangent are indispensable. Since these BASIC platforms use radians instead of degrees, though, I have written three functions for the graphics template that handle degree values, sind, cosd, and tand. Use them as you would sin, cos, and tan, but you may simply do your thinking in degrees. It may be easier working with 360 degrees than 6.28-odd radians for a circle.

The bmpw and bmph functions allow you to instantly determine the width or height, respectively, of a bitmap image file. Bmpw("hello") returns the width of "hello.bmp". The function is designed to handle the file name with or without the ".bmp".

Using this graphics template makes programming blind a lot easier. You may wish to change the name of the window from "lab" to whatever you want.

Two skills you may be gathering you need are the abilities to handle math and bitmap coding, the former being most important. Since every pixel on the screen has XY coordinates, use of algebra, geometry, and some basic trigonometry, can enable a totally blind person to do what most sighted people think is impossible: draw pictures. If you draw a circle over and over, having the center the same, but the radius increase as you go, you will make a spiral, for example. I also found learning about parametric equations on Google helpful. With parametric equations sets, one equation is for the X value, and the other equation is for the Y value. The equations also contain a T variable which changes in an orderly way as you run the equations, and, in so doing, you draw something. It may seem tricky at first, but if you really wish to draw detailed pictures, it's worth it and possible. I used these concepts to code a 3-minute, animated movie in JB.

Looking over the user functions in the graphics template will show you some about how bitmap files are coded. Browsing Google shows even more. You can actually write programs that have the computer re-code bitmap files to, for example, change all the red to green, or remove unneeded black around an image. This is more advanced, but very useful. Appendix B contains code for a sprite maker. Sprites are specially-designed graphics one uses to make animations. The background around the image is invisible, so a horse would just be seen galloping across a field. This program takes a bitmap drawn against a solid "0 0 0" black background and creates the white, silhouette "mask" that the sprite needs to be a successful animation.

Some other brief notes about coding graphics follow. I usually use the "flush" command a little more than most programmers since I cannot see if part of a picture I'm coding might disappear. I also use the "down" command in about every line of image drawing just to make sure the "pen" is, indeed, on the screen, drawing. I also have to remind myself to put the "drawsprites" command in when animating something as that's the only way the animation will actually appear. I've had enough experiences with my wife saying, "I don't see anything but a white screen." To instill this in my mind. I also seldom draw entire pictures in one sitting and with one program. I usually make all my images first, turn them into sprites, and then put them on a background I've prepared. This way, it is easier to make modifications according to advice from sighted friends, and, by the way, sighted people should get second opinions of graphic art work they make, as well.

Sometimes the "drawsprites" and "fill" commands are more delayed when using a screen reader. (The "fill" command fills an entire graphics window with whatever color is chosen.) It may be better, then, to run animations for sighted viewers, with the screen reader off. You can also set up loops for monitoring the time a sprite stays before moving again, and have the loop keep issuing a "drawsprites" command until the desired time runs out. As far as the "fill" command goes, for drawing images, just use col$ in my graphics template to check a part of the screen you know should be the desired background color when you run the program.

I should say something here about coding text into a graphics window. If you're not using LB with the "stringwidth?" command, you may need to use other ways to find out how wide a string of characters is, especially if you want to make a bitmap of it for spriting. You can specify the font type you want by number of pixels rather than point size. For this, you end the "font" command with two numbers, not one. "font 30 40" will produce type that averages 30 pixels across and 40 pixels high. The "posxy" command will tell you the number of pixels between the row the text is on and the next row, but you'll just have to guess the width. You can change the window width, also, to be as wide as you think the text should be based on the pixel count and see if it still fits. Your screen reader should have hot keys on it that move the mouse pointer, and typed text like this will show up and be recognized as readable.

I'd like to mention, briefly, some of the most useful graphics commands for a blind person. The "posxy" command tells you the precise coordinates of where you are when drawing an image. If you've told the computer to turn so many degrees and go a certain number of pixels, this command will enable you to see if that actually happened. There are also easy commands for drawing lines, circles, ellipses, boxes, and pie portions. The "stringwidth?" command, only in LB and RB, tells precisely how many pixels long a piece of text is you wish to put in a window. The "spritecollides" command tells if an animation aimage is overlapping another one, and, if so, which ones it overlaps. The "spritexy?" command tells precisely where an animation is on the screen. It should be noted that while the "spritescale" command will shrink or blow up an animation image, shrunk images can contain a few black dots, and blown-up images can be hazy around the edges. Using this command, though, can still be helpful. No one is expecting George Lucas graphics from a blind person.

Finally, with graphics, be sure—and I must emphasize this—to be clear in your naming of your image files. Since you are blind, that file name is all you have to know what the picture is. A name like "ManStandingRightArmUp.bmp" is perfectly fine for a blind user.

DLLs

LB's ability to make DLL and API calls is very powerful, though it does require some advanced coding skills. These skills can be acquired, though, with a little patience and an adventuresome heart. Since text books on the subject would be inaccurate after scanning, one must find digital sources of information.

I started by examining guides about DLLS for LB, the Help files, and sample programs like Piano 6. I'd copy and paste the code for various MIDI functions into a MIDI template I wrote and figured out rather quickly how to change notes and tones. Then I went outside the LB universe to general Microsoft web sites. I compared DLL calls there in C++ with LB commands for the same DLL in LB programs. This way I was able to learn how to translate between the two systems. "INT," in C++, for example, is like "lONG," in LB. "UINT," in C++, is like "ULONG," in LB. Then, I was able to explore DLLs that LB program samples don't discuss. I used what I'd learned from C++ and LB here to determine how to set up LB calls and when and how to use structs. I was able to draw circle arcs and other portions of circles difficult to manually code.

Appendix F contains links to some web sites helpful for learning to work with DLLs. The first entry, there, is how to make the computer beep, and I even provide LB equivalent code. It's very simple. There is also a link to an LB dll for speech which comes with sample code. Text to speech with LB is possible.

Run BASIC

As far as Run BASIC goes, the coding aspect is no less accessible. You may need a sighted friend, though, to help you publish the web pages you make and to help with eye appeal.

This brief article doesn't say everything a blind person would need to write programs, but it should get one off to a good start. Now, I present the appendix programs that you can just run to get going well. You may also wish to check out "BASIC Rules" with the article on BASIC programming for blind users. This program is a blind-accessible graphics analyzer which helps one explore the "rule" command for graphics windows. One setting in this program allows part of the screen to become any RGB color. This is good for having a sighted friend help you choose just the best color for something.

Appendix A

Graphics Template

Code follows:

```
WindowWidth = DisplayWidth
WindowHeight = DisplayHeight
open "Lab" for graphics as #g
#g "trapclose [quit]"
[work]

wait
[quit]
close #g
end
'Function col$ -- col$(x,y) finds RGB value for pixel x,y.
function col$(colorx,colory)
#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
'Function sind -- sind(x) finds the sine of x in degrees.
function sind(trigx)
sind = sin((3.141592653590*trigx)/180)
end function
'Function cosd -- cosd(x) finds the cosine of x in degrees.

function cosd(trigx)
cosd = cos((3.141592653590*trigx)/180)
end function
'Function tand -- tand(x) finds the tangent of x in degrees.
function tand(trigx)
tand = tan((3.141592653590*trigx)/180)
end function
'Function bmpw -- bmpw("bitmap") gives the width of that bitmap.
function bmpw(bmpname$)
if upper$(right$(bmpname$,4)) <> ".BMP" then bmpname$ =
 bmpname$ + ".bmp"
open bmpname$ for input as #fileread
bmpinfo$ = input$(#fileread, lof(#fileread))
close #fileread
bmpw = asc(mid$(bmpinfo$, 19, 1)) + asc(mid$(bmpinfo$, 20, 1)) * 256
end function
'Function bmph -- bmph("bitmap") gives the height of that bitmap.
function bmph(bmpname$)
if upper$(right$(bmpname$,4)) <> ".BMP" then bmpname$ =
 bmpname$ + ".bmp"
open bmpname$ for input as #fileread
bmpinfo$ = input$(#fileread, lof(#fileread))
close #fileread
bmph = asc(mid$(bmpinfo$, 23, 1)) + asc(mid$(bmpinfo$, 24, 1)) * 256
end function
```

Appendix B

Sprite Maker

Code Follows: (If your file is not a 32-bit bitmap, the program will open a graphics window and save it as one before making the sprite.)

```
cls
print"Sprite Maker"
print "Convert 32-bit bitmap files into JB/LB sprites."
filedialog "Enter source bitmap name.", "*.bmp", a$
lf = len(a$)
lg = lf
do
lf = lf - 1
loop until (mid$(a$,lf,1) = "\") or (mid$(a$,lf,1) = "")
fa$ = left$(a$,lf)
fb$ = mid$(a$,lf+1,lg-lf-4)
fc$ = fb$+"Spr.bmp"
filedialog "Enter sprite name.", fc$, b$
if (upper$(right$(b$,4)) <> ".BMP") then
out$ = b$ + ".bmp"
else
out$ = b$
end if
print
print "Please wait."
open a$ for input as #f
s$ = input$(#f,lof(#f))
t = lof(#f)
close #f
[working]
if asc(mid$(s$,29,1)) <> 32 then goto [convert]
w = bmpw(a$)
h = bmph(a$)
dim q$(1+int(t/10000))
dim a(t)
dim b(t)
for k = 1 to t
a(k) = asc(mid$(s$,k,1))
next k
a(3) = (a(3) * 2)-66
a(4) = a(4) * 2
a(5) = a(5) * 2
if a(3) > 255 then
a(3) = a(3) - 256
a(4) = a(4) +1
end if
if a(3) < 0 then
a(3) = a(3) + 256
a(4) = a(4) - 1
end if
if a(4) > 255 then
```

```
a(4) = a(4) - 256
a(5) = a(5) + 1
end if
if a(4) < 0 then
a(4) = a(4) + 256
a(5) = a(5) - 1
end if
a(35) = a(35) * 2
a(36) = a(36) * 2
a(37) = a(37) * 2
if a(35) > 255 then
a(35) = a(35) - 256
a(36) = a(36) + 1
end if
if a(36) > 255 then
a(36) = a(36) - 256
a(37) = a(37) + 1
end if
a(23) = a(23) * 2
a(24) = a(24) * 2
if a(23) > 255 then
a(23) = a(23) - 256
a(24) = a(24) + 1
end if
f = w*h
for x = 1 to f
m = 63 + 4*x
if a(m) = 0 and a(m+1) = 0 and a(m+2) = 0 then
b(m) = 255
b(m+1) = 255
b(m+2) = 255
else
b(m) = 0
b(m+1) = 0
b(m+2) = 0
end if
next x
c$ = ""
for g = 1 to 66
c$ = c$ + chr$(a(g))
next g
c$ = c$ + right$(s$,t-66)
r = 1
u = 0
for g = 67 to t
u = u + 1
```

```
if u > 10000 then
u = 1
r =r + 1
end if
q$(r) = q$(r) + chr$(b(g))
next g
open out$ for output as #f
print #f, c$;
for i = 1 to r
print #f, q$(i);
next i
close #f
playwave "nofile.wav", async
print "Done."
                    end
[convert]
loadbmp "bit", a$
WindowWidth = DisplayWidth
WindowHeight = DisplayHeight
open "Convert to 32-Bit BMP" for graphics as #g
#g "trapclose [Quit]"
w = bmpw(a$)
h = bmph(a$)
#g
"place
0 0 ; down ; Color Black ; BackColor Black ; BoxFilled ";w+10;" ";h+10
#g "drawbmp bit 1 1"
#g "flush"
for k = 1 to 50000
next k
#g "getbmp bitnew 1 1 ";w;" ";h
bmpsave "bitnew", "temp.bmp"
check = 1
print #g, "flush"
[Quit]
close #g
if check = 0 then goto [convert]
open "temp.bmp" for input as #f
s$ = input$(#f,lof(#f))
t = lof(#f)
close #f
                                kill "temp.bmp"
goto [working]
[functions]
function bmpw(f$)
open f$ for input as #f
```

```
s$ = input$(#f, lof(#f))
close #f
bmpw = asc(mid$(s$, 19, 1)) + asc(mid$(s$, 20, 1)) * 256
end function
function bmph(f$)
open f$ for input as #f
s$ = input$(#f, lof(#f))
close #f
bmph = asc(mid$(s$, 23, 1)) + asc(mid$(s$, 24, 1)) * 256
end function
```

Appendix C

Here, we're going to draw a heart. I figured out a parametric formula for doing such and base this program on it. If the math is too complex for where you are mathematically, the program will still run. You'll just be able to make hearts without knowing exactly why, and that's fine. There isn't space here to adequately explain the math. What I do here, though, is help you understand how to use the program so you can make nice heart images for that special someone in your life and how to use the col$ to read pixel color.

This program draws hearts with emphasis on the point "666,206" which is the point where the two half-circles on top come together. (The center pixel of the screen seems to be (666,349) when all is said and done. Below that point, the outer lines slowly straighten until they join in a point somewhere below, depending on the top circle radius and the bottom vertex/point angle. (Remember, the radius is the distance from the outer edge of a circle to the center.) At least, this is how I code hearts.

When you look at the program, first, you'll find a list of variables and what they equal. If I'm writing a program and wish to experiment with different variable values and don't want to code a menu or list of choices to go through at the beginning, I'll just do this. I can change any value very quickly without searching through the program, since I don't have eyes that scroll easily.

If you use the default values I have entered, you'll run the program and make a filled heart that is red and has a top circle radius of 50 pixels. Since the top of the heart is made of two half-circles, the heart is 200 pixels across. The bottom vertex angle is 90 degrees. This heart, then, ends up being 200 pixels wide by 170 pixels high. I coded the program to print out this type of information in the main window. Alt-tab over to read it after the program runs, or use alt-f4 to close the window. You can browse through the information with the cursor, learning the dimensions of the heart and its coordinates. If you change the radius and vertex angle values and the bottom of the heart is below a Y-value of about 694, the heart may not all fit on the screen for saving as a bitmap. If you set the value of "save" to 1, the file will be saved with whatever is on the screen. A file that goes down too far may pick up some of the designs on the screen below the graphics window. If the vertex angle is 90 degrees, a heart with top circle radius 200, (800 pixels across) should fit fine for large size.

Now, let's work with col$. First, the col$ function only works in programs with this template copied in and that have #g as the handle for the graphics window. If you wish for this function to work in a situation with a different handle, you will need to edit the function accordingly.

If you search the Heart Maker program, you'll find a branch label called [research], and, just before that is a notation that this is a good place for col$ work. Just search forward for "a good place" without the quotes to get there, or use alt-G in LB to jump right to the place.

Type the following code, or copy-paste it, in this [research] branch section of the Heart Maker program:

```
print col$(666,240)
print col$(1100,500)
```

Run the program. The first point is inside the filled heart. The second point is outside the filled heart. After all the information about heart size, the main window will show 2 sets of 3 numbers. The first one says "255 0 0" and that is the code for red, where red is 255, green is 0, and blue is 0. The second set of numbers is "0 0 0" where all numbers are 0. That is the code for black. Try playing around with the color values/names for heartColor$ and bColor$ (the background color line) and see what happens to the RGB values.

Next, change the value of "filled" at the top of the program so that it equals 0. Set the colors for the heart and the background to values where you know the RGB values for them. Delete the two lines of code you typed about printing col$ and replace them with the following 3 lines:

```
for check = 1 to 1100
print check ; "  " ; col$(check,250)
next check
```

When you run the program, it will take a moment to check all those pixels, but when done, you'll have a long list of 1100 RGB values with the X-value of the pixel just before them. Use ctrl-f to search forward for the RGB value you selected for the heart color, putting just one space between the numbers. The cursor will end up right on the pixel information about that pixel where the hollow heart begins. You can read/search down to learn more about that row of pixels. If you're wanting to do serious graphics coding, these tricks will become very important.

Now, here's the code. Enjoy playing with the radius and angle values, making hearts for that special someone in your life. Yes, now there is a formula for love.

Code follows:

```
'Heart Maker
'A parametric formula concept used to draw a heart.

'Set top circle radius, up to 200 if vertex angle is 90.
radius = 50
'Set bottom vertex angle.
```

```
angle = 90
'Filled or hollow. 1 is filled; 0 is hollow
filled = 1
'Set background color, with  simple color name or RGB.
bColor$ = "Black"
'Set heart color, by name or RGB. Raising last two numbers moves from
red to pink to white.
heartColor$ = "255 0 0"
'Save or not. 0 means the image won't be saved.
save = 0
'Choose file name to save as, leaving off ".bmp".
fileName$ = ""
'Open graphics window.
WindowWidth = DisplayWidth
WindowHeight = DisplayHeight
open "Lab" for graphics as #g
#g "trapclose [quit]"
[work]
'Draw background.
#g "Fill ";bColor$
#g "flush"
'Preliminary math short cuts.
g = (360-angle)/4
h = 360 - g
i = 360 - (2*g)
[HeartDrawingLoop]
 for t = 0 to h step 0.1
 u = radius / cosd((abs(t-i) + (t-i))/2)
a = u * cosd(t) - radius
b = (u * cosd(t) - radius)*-1
c = u * sind(t)
#g "Color ";heartColor$
xa = int(a + 666)
xb = int(b + 666)
y =  int((c*-1) + 206)
#g "size 2 ; down ; set ";xa;" ";y
#g "down ; set ";xb;" ";y
'Optional filling.
if filled = 1 and t < 180 then
#g "size 2 ; down ; line ";666-radius;" ";y;" ";xa;" ";y
#g "line ";666+radius;" ";y;" ";xb;" ";y
end if
if filled = 1 and t >= 180 then #g
"size 2 ; down ; line ";xa;" ";y;" ";xb;" ";y
 next t
#g "flush"
```

```
[bitmapBoundaries]
n = 666 - (2*radius) - 2
o = 206 - radius - 2
p = radius * 4 + 4
q = 2 + y - o
#g "getbmp h ";n;" ";o;" ";p;" ";q
if save = 1 then bmpsave "h", filename$+".bmp"
'Print heart size.
print "This heart is " ; p-4 ; " pixels wide by " ; q-4 ;
" pixels high."
print
"It has upper left coordinates, ";n+2; ", ";o+2;
" and lower right, ";n+p-3;", ";o+q-3;"."
'This is a good place for col$ work.
[research]

wait
[quit]
close #g
end
'Function col$ -- col$(x,y) finds RGB value for pixel x,y.
function col$(colorx,colory)
#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
'Function sind -- sind(x) finds the sine of x in degrees.
function sind(trigx)
sind = sin((3.141592653590*trigx)/180)
end function
'Function cosd -- cosd(x) finds the cosine of x in degrees.

function cosd(trigx)
cosd = cos((3.141592653590*trigx)/180)
end function
'Function tand -- tand(x) finds the tangent of x in degrees.
function tand(trigx)
tand = tan((3.141592653590*trigx)/180)
```

```
end function
'Function bmpw -- bmpw("bitmap") gives the width of that bitmap.
function bmpw(bmpname$)
if upper$(right$(bmpname$,4)) <> ".BMP" then bmpname$ =
 bmpname$ + ".bmp"
open bmpname$ for input as #fileread
bmpinfo$ = input$(#fileread, lof(#fileread))
close #fileread
bmpw = asc(mid$(bmpinfo$, 19, 1)) + asc(mid$(bmpinfo$, 20, 1)) * 256
end function
'Function bmph -- bmph("bitmap") gives the height of that bitmap.
function bmph(bmpname$)
if upper$(right$(bmpname$,4)) <> ".BMP" then bmpname$ =
 bmpname$ + ".bmp"
open bmpname$ for input as #fileread
bmpinfo$ = input$(#fileread, lof(#fileread))
close #fileread
bmph = asc(mid$(bmpinfo$, 23, 1)) + asc(mid$(bmpinfo$, 24, 1)) * 256
end function
```

Appendix D

A Simple Sprite Background

Here's a simple piece of code you can insert in the [work] section of the graphics template to make a solid blue background to put heart sprites on:

Code follows:

```
#g "Fill Blue"
#g "Flush"
print col$(1200,500)
#g "getbmp a 0 0 1333 698"
bmpsave "a", "Blue.bmp"
```

The Blue.bmp bitmap will be saved, and the pixel requested should be "0 0 255" for RGB values, red 0, green 0, blue 255. That way you know the background was made properly.

Appendix E

Homework

Make a few heart images of various configurations. Make sprites of them. Look over some of the material on sprite commands online or in the help/tutorial section of the BASIC platform you have. Then, write a program for someone you love in which the hearts you made move on the screen. You can do it. I did it before I had all these fancy tools. Have fun.

Appendix F.

Here are some helpful links to get one started on a journey of using DLLs and APIs in LB. They are arranged in order of training use. Going through them in this order ensures best learning.

"Beep function (Windows)." https://msdn.microsoft.com/en-us/library/windows/desktop/ms679277(v=vs.85).aspx
You'll find here, information in C++ on how to make the computer sound a simple beep of your choice of frequency and duration. You can see how this works very easily. Here is the code for doing this in LB.

```
calldll #kernel32, "Beep", frequency as long, duration as long,
result as boolean
```

Play around with this, setting different values for "frequency" and "duration." Frequencies are in hertz and durations are in milliseconds.

http://basic.wikispaces.com/SpeechLibrary
This is the link to a speech dll in LB. The computer will speak a string you enter.

Moore, Brad. "The beginner's guide to API and DLL calls."
http://www.libertybasicuniversity.com/lbnews/nl101/5.htm
This article contains general information about DLLs, APIs, and structs.

Alyce. Liberty BASIC Programmer's Encyc - FloodFill. http://lbpe.wikispaces.com/FloodFill
This article explains how to use the very practical DLL, "FloodFill" so a blind person need only draw the border of an image and position himself/herself somewhere inside to color it in. You do not need to use structs for this, so it is more straightforward for beginners.

https://msdn.microsoft.com/en-us/library/windows/desktop/dd162709(v=vs.85).aspx
"ExtFloodFill Function." Here is an article for how to use FloodFill in C++. You can compare the code to the LB code in the article above to learn how to read these types of C++ articles.

Alyse. "Liberty BASIC Programmer's Encyc - APIPolygon." https://lbpe.wikispaces.com/APIPolygon
This site shows how to code polygons using DLL commands and structs. It is very clear, in this article, how to work with structs.

"Windows GDI (Windows)" C:\Program Files (x86)\Liberty BASIC Pro v4.04\lb4help\LibertyBASIC_4_web\html\libe4yur.htm
There are all kinds of DLLs for graphics here that one can use in LB that can make graphics much doable and more detailed for blind programmers. These are all in C++, but it is not difficult to figure out how to use them in LB with a little practice.

There are also keyboard commands on these Microcontrol DLLs on these Microsoft web sites. Google DLLs like "GetAsyncKey" and compare what you find with LB's "Key Press" program which uses the same DLL.

Basic Programming for Blind Programmers | General Information | Graphics Coding | Run BASIC | Appendix A | Appendix B | Appendix C | Appendix D | Appendix E | Appendix F.