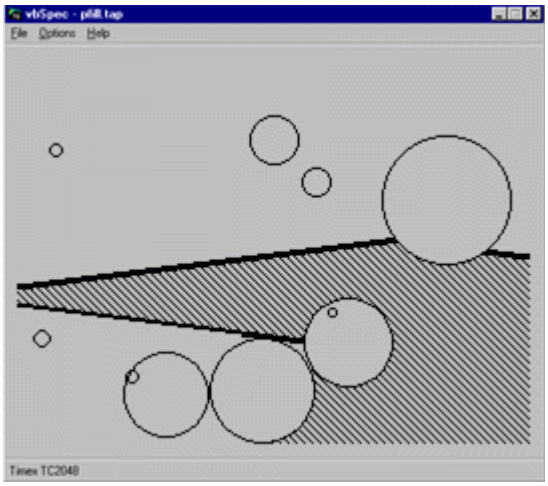
A Fast Well-Behaved Pattern Flood Fill
Alvin Albrecht



Sinclair Basic comes with a number of graphics primitives that are easy to use but they only manage to scratch the surface when it comes to computer graphics. There are many interesting graphic tools that are practically begging to be implemented on the ts2068, the flood fill being one of them. But before we can tackle the subject of flood fills, a small review of recursion is in order.

A Review of Recursion

In plain terms a recursive subroutine is a subroutine that calls itself. A typical recursive algorithm breaks a complicated problem into simpler pieces, then applies itself to those pieces repeatedly until the original problem is broken into many tiny problems that can be trivially solved. I have written about recursion in a past issue of ZQA so I will not be rehashing that material here. Instead, as a refresher, let's investigate the one recursive algorithm that everyone sees in Computer Science 101 -- the computation of a factorial.

N! (N factorial) is defined as (N)x(N-1)x(N-2)x…x1 with N being a positive integer. 5! = 5x4x3x2x1 = 120, for example. 0! is defined as 1. A recursive solution might look like this:

```
int factorial(int n)
{
   if (n <= 1)               // if n<1 the answer is 1
      return 1;
   return n*factorial(n-1);  // else break into smaller problem
}
```

This is a C function that computes N! but don't let that put you off - C is a fairly easy language to understand. Sinclair Basic does not fully support recursion so a Basic version would require artifacts surrounding it, which would only obscure

the point I want to make. Now that a decent C compiler is available for our T/S machines, I don't feel too guilty about sticking with the C.

We make a call, asking to compute the factorial of N. If N <= 1 the answer is 1. Otherwise the problem is too difficult to solve so it is broken into a smaller one: N times the factorial of (N-1).

That was easy, but it may be initially surprising to learn that this is a poor method for calculating factorials. To understand why, we'll need to pay closer attention to what is involved in a recursive call.

The factorial function, as written above, needs to remember two things: the number N and where it was called from so that it can return there later. Compiling the above for a ts2068 using the z88dk C compiler would reserve two bytes to store N and two bytes for the return address, for a total of four bytes. The "return" value is passed in the z80's HL register pair which is free. So the initial call to compute 5! would require four bytes to be reserved on the stack. But that is not the end of the story. Factorial(5) would try to return "4*factorial(4)", causing another call to be made to factorial with N=4, requiring another four bytes. Factorial(4) would make a call to factorial(3), requiring a further four bytes, etc. In other words, to find the answer to factorial(5), the computer would make calls to factorial(5), factorial(4), factorial(3), factorial(2) and factorial(1). We say that factorial(5) has a recursion depth of 5 because a maximum of five instances of factorial will exist at any one time during its computation. With each instance needing four bytes to remember its value of N and return address, factorial(5) requires 5*4=20 bytes of memory to compute its result. Generalizing, we can say that the recursion depth of factorial(N) is N and 4*N bytes are required to compute the result.

Even on our small 64K machines that doesn't seem to be a lot of memory and really it isn't. Things get worse if you try to compute something like 69! which would seem to require 276 bytes in the recursive solution. I say "seem to" because we would actually need to introduce a new large variable to hold the result in each recursive step -- 69! requires 41 bytes to hold its value precisely! Switching to a floating point representation would reduce that to four bytes (at the expense of precision), compared to the two bytes we've assumed (the result in each step is held in the HL register pair, a consequence of how the z88dk C compiler does things). With the way the subroutine is written now we couldn't correctly compute anything more than 8! and therefore memory usage is never really an issue. Other recursive algorithms may have a recursion depth in the thousands with dozens of bytes needed for each instance. Then the matter of memory is significant, and, indeed, you will see one such example shortly.

Another problem with the factorial recursive solution is runtime. It takes time to set up calls and return from them -- these actions translate directly into pushes

and pops on the z80's stack. Compare this recursive solution to the alternative iterative solution below:

```
int factorial(int n)
{
   int fact, i;

   fact = 1;              // the answer starts at 1
   for (i=2; i<=n; i++)   // i=2, while i<=n execute the loop
      fact = fact*i;
   return fact;
}
```

We need two bytes for "fact", two bytes for "i", two bytes for "N" and two bytes for the return address = 8 bytes total no matter what "N" is. There are no calls to set up, just a for loop. This version will be faster and use up a small, fixed and predictable amount of memory. It is superior to the recursive solution in every way.

So what is the conclusion of all this discussion? Recursion must be used with care. It can be a panacea to solve many very difficult problems, but you must be fully aware of how much memory will be required and the runtime necessary in comparison to an equivalent iterative solution. The Towers of Hanoi solution in a back issue of ZQA has a maximum recursion depth of 64 (for 64 disks) and the Knight's Tour has a recursion depth of 64 (the number of squares on a chess board), very manageable numbers.

A Recursive Flood Fill Algorithm

So what has all this got to do with flood filling? It turns out that the obvious approach to filling an arbitrary region involves a recursive solution. And the recursive solution is a bad one.

In comp.sys.sinclair, Geoff Wearmouth shared a Basic subroutine from an early '80s type-in magazine that would fill an arbitrary area on screen bounded by a solid pixel boundary. Here it is:

```
 5 REM AUTHOR UNKNOWN
10 CIRCLE 128,88,80
20 LET x=100 : LET y = 100 : REM START POINT
30 GO SUB 1000 : STOP

1000 PLOT x,y
1010 IF NOT POINT(x+1,y) THEN LET x=x+1 : GO SUB 1000 : LET x=x-1
1020 IF NOT POINT(x-1,y) THEN LET x=x-1 : GO SUB 1000 : LET x=x+1
1030 IF NOT POINT(x,y+1) THEN LET y=y+1 : GO SUB 1000 : LET y=y-1
1040 IF NOT POINT(x,y-1) THEN LET y=y-1 : GO SUB 1000 : LET y=y+1
1050 RETURN
```

The main subroutine begins at line 1000 and the algorithm used is a recursive one, the evidence being the "GOSUB 1000" statements in the subroutine itself. The fill subroutine above plots the current point and then tries to move in all four directions away from the point. Before each move it checks to see if the point is already black, indicating a boundary. If not, it is considered a valid move and a fill is initiated from that point by another recursive call to line 1000 with the new pixel coordinate in (x,y).

Earlier I mentioned that Sinclair Basic does not fully support recursion. The reason it doesn't can be seen in this fill program. Each run through the subroutine at 1000 expects to have its own private copy of (x,y). The value of (x,y) at 1010 must be the same value at lines 1020, 1030 and 1040 in order for the program to work. But there are one or more "GOSUB 1000" calls in the middle, which themselves require new values of (x,y) and which will themselves change (x,y). Sinclair Basic has only one copy of these variables which must be shared by each recursive call. A recursive C call would give each "GOSUB" a private copy of (x,y) on the stack independent of all other "GOSUBs". Not so in Sinclair Basic. So the problem is, after each "GOSUB 1000" in the fill subroutine, how do we make sure that our own (x,y) has not changed? In the above code, the solution is simple. We promise that when "GOSUB 1000" returns, the value of (x,y) is not changed from what it was when "GOSUB 1000" was executed. In line 1010, for example, x is increased by one before a call to "GOSUB 1000". Because of our promise (the jargon calls such a promise an "invariant") we know that when the GOSUB returns, x will be one larger than what it was at the beginning of line 1010. So to get x back to where it was, decrease by one and everything will be fine for the next line. Before the routine returns in line 1050 we know that (x,y) has not changed from its initial state in line 1000. That's the subroutine keeping its promise.

This fill algorithm is called a flood fill because the fill "floods away" from the initial point in all directions. Other fill algorithms exist, but this one is both easy to understand and capable of filling any arbitrary region without restrictions. Earlier I hinted that a recursive solution to the flood fill problem is a bad one. I'll leave that thought here and come back to it later when we've looked at a couple of machine code implementations of the algorithm. For now, realize that each "GOSUB" requires the ts2068 to remember a return line number (two bytes) and then consider what the recursion depth might be for a 256x192 resolution blank screen (hint - you wouldn't be far off if you just multiplied 256 and 192 together!).

If you typed in the Basic program and ran it, you'd realize that it is mighty slow. Any useful fill utility will need to be written in machine code. To do that, we will need to review the structure of the ts2068's display file.

Display File Organization

The ts2068's display file is where all the screen information is stored. The SCLD chip constructs the TV display by reading the information stored there. The display file is "memory-mapped" because the storage exists in the z80's memory space, from address 16384 to 22527. If you poke values into those addresses you will see the display change. In the ts2068's other display modes (dual screen, hi-colour, hi-res) more areas of memory are used to hold the display. In this article, we'll only concern ourselves with the default 256x192 mode.

A pixel display occupying 16384 to 22527 reserves 6144 bytes to store all the screen information. The ts2068 has a resolution of 256*192 = 49152 pixels. How do we cram information about 49152 pixels into 6144 bytes? Well, each pixel can be represented by one bit - either one or zero, on or off. Cramming 8 pixels into a byte, we'd need 256*192/8 = 6144 bytes. Problem solved!

A simple way to organize the display might have pixels 0..7 for the top line of the display stored at address 16384, pixels 8..15 at address 16385, … pixels 248-255 stored at address 16415. The next pixel line would follow with pixels 0..7 of line 1 at address 16416, and so forth for all 192 lines on the screen. This is indeed how the TV draws its display, left to right, top to bottom. But the display organization was chosen to optimize the printing of characters so it's not done in this simple manner. To see evidence of this, try this short program:

```
10 FOR z=16384 TO 22527
20 POKE z,255
30 NEXT z
```

On the largest scale you will notice that the display is divided into three parts called blocks. First the top block is filled, then the second and finally the third. Each block is further divided into eight character lines. Each of these lines is divided into eight scan lines. The first scan line for all character lines in a block is filled, followed by the second scan line for all character lines, and so on to the final eighth scan line. Each scan line itself is composed of 32 horizontal bytes with each byte holding eight pixels.

This organization sounds complicated but it really isn't that bad if some thought is applied to it. By paying attention to how the display is built up in increasing byte order, we can construct a screen address given block, character line, scan line and column as follows:

FIGURE 1. Screen Address Organization in Binary

0	1	0	B	B	S	S	S	L	L	L	C	C	C	C	C

Where:

BB = screen block, 0..2
SSS = scan line, 0..7
LLL = character line, 0..7

CCCCC = horizontal byte / character, 0..31

While observing the Basic program in action, you'll notice that the horizontal column changes the fastest. There are 32 columns, requiring 5 bits to represent those. They increase the fastest so they appear in the bottom 5 bits of the 16-bit address. The next fastest thing that changes is the character line. There are 8 lines in each block, requiring 3 bits to represent them. These 3 bits appear next to the column bits. Next, in order of fastest changing, are the scan lines (8 of them requiring 3 bits) followed by the block (3 of them requiring 2 bits). The display starts at address 16384 (0x4000) so we add that to our 16-bit address. This is responsible for the lone '1' you see in figure 1.

The character position row = 10, column = 12 is located in block 1 (the second block since it holds the second third of the display, rows 8..15), line 2 (the third character line in this block -- rows 8, 9, **10**), scan lines 0 (top) through 7 (bottom) for the full character square, and column 12. This leads to a screen address that looks like:

0	1	0	0	1	S	S	S	0	1	0	0	1	1	0	0

With various values of SSS:

SSS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Screen	484C	494C	4A4C	4B4C	4C4C	4D4C	4E4C	4F4C
Address	18508	18764	19020	19276	19532	19788	20044	20300

To print a letter 'A' at (10,12), poke the appropriate values into memory at these addresses:

```
POKE 18508,BIN 00000000
POKE 18764,BIN 00111100
POKE 19020,BIN 01000010
POKE 19276,BIN 01000010
POKE 19532,BIN 01111110
POKE 19788,BIN 01000010
POKE 20044,BIN 01000010
POKE 20300,BIN 00000000
```

At this point, you may realize why UDGs and printed characters are 8x8 pixels in size. There are 8 vertical scan lines in each character line and there are 8 pixels packed into a byte. But you may not realize why this particular display file organization speeds up character printing. If you back up and look at the screen addresses computed above, you'll notice that each scan line is separated by exactly 256 bytes. In assembly language, an address is held in a register pair, like HL. Adding 256 to an address to move to the next scan line is a simple matter of incrementing the most significant register, in this case H with the "INC H" instruction. That's all it takes! Moving horizontally to the right one character position involves adding one to the screen address (ie adding one to "CCCCC" in

figure 1), which can be done just as quickly with "INC L". You can't get any faster than that. In fact, this display file organization was patented by Sinclair's Richard Altwasser back in 1982 (visit http://wearmouth.demon.co.uk/ to see the patent).

That's all fine and good but we still haven't managed to easily map a pixel coordinate to a screen address. Here's how we do it:

FIGURE 2. Mapping Pixel Coordinates to Screen Address Units

X								Y							
C	C	C	C	C	T	T	T	B	B	L	L	L	S	S	S

The thought process that led to figure 2 is similar to the previous one. The X coordinate is more or less obvious: there are 32 columns horizontally (5 bits) with each column containing 8 pixels (requiring 3 bits). The pixel position within a byte (0..7) changes fastest as we move horizontally so it appears as "TTT" in the least significant bits of X. For the Y coordinate, the fastest changing items as we move from the top of the screen to the bottom are the scan line, followed by the character line, followed by the block.

Given an X coordinate in the range 0-255 and a Y coordinate in the range 0-192, convert them to binary as in figure 2 and reassemble the bits as in figure 1. For example, pixel coordinate (x,y) = (133,67) in binary is (1000 0101, 0100 0011) with CCCCC=10000, BB=01, LLL=000, SSS=011 according to figure 2. Moving the bits around to the form in figure 1 gives an address of "0100 1011 0001 0000" or 19216 in decimal. The bits "TTT" in the X coordinate do not appear in figure 1. They identify which bit within the screen byte corresponds to the individual pixel. "0" corresponds to the leftmost bit and "7" corresponds to the rightmost; in this case it's 5. To plot the pixel (133,67) we could simply "POKE 19216,BIN 00000100" where the single '1' in the BIN statement sits in bit 5 from the left. Keep in mind that the pixel coordinates I am using have the screen's origin located at the top left corner of the screen. This is different from TS2068 Basic which places the origin 16 pixels above the bottom left corner of the screen.

If this procedure had to be done by hand for each pixel, it would get tedious quickly. Here's a short machine code routine that does it for us:

```
; Get Screen Address
;
; Returns the screen address and pixel mask corresponding
; to a given pixel coordinate.
;
; enter: a = h = y coord
;        l = x coord
; exit : de = screen address, b = pixel mask
; uses : af, b, de, hl
```

```
.SPGetScrnAddr
   and $07     ; A = 00000SSS
   or $40      ; A = 01000SSS
   ld d,a      ; D = 01000SSS
   ld a,h      ; A = Y coord = BBLLLSSS
   rra
   rra
   rra         ; A = ???BBLLL
   and $18     ; A = 000BB000
   or d        ; A = 010BBSSS
   ld d,a      ; D = 010BBSSS top 8 bits of address done

   ld a,l      ; A = X coord = CCCCCTTT
   and $07     ; A = 00000TTT
   ld b,a      ; B = 00000TTT = which pixel?
   ld a,$80    ; A = 10000000
   jr z, norotate   ; if B=0, A is the right pixel so skip

.rotloop
   rra         ; rotate the pixel right one place B times
   djnz rotloop

.norotate
   ld b,a      ; B = pixel mask
   srl l
   srl l
   srl l       ; L = 000CCCCC
   ld a,h      ; A = Y coord = BBLLLSSS
   rla
   rla         ; A = LLLSSS??
   and $e0     ; A = LLL00000
   or l        ; A = LLLCCCCC
   ld e,a      ; E = LLLCCCCC
   ret         ; DE = 010BBSS LLLCCCCC, the screen address!
```

The subroutine is called with A=H=Y coordinate and L=X coordinate and we get the screen address in DE and the pixel mask in B on the way out. If we ORed B into (DE), we could plot the pixel. If we ANDed the complement of B into (DE), we could unplot the pixel and if we ANDed B with (DE) we could test whether the pixel was set.

This subroutine is great for calculating a screen address corresponding to a pixel position from scratch, but you'll notice that it is rather lengthy and therefore slow, in a relative sense. Frequently you'll be plotting a pixel and then plotting many more nearby, possibly a single pixel away. For example, in the process of drawing a line, the initial point is plotted and then succeeding points above, below, to the left or right are plotted. We could handle the drawing of the line as plotting many individual pixel points, calling the above subroutine to compute the screen address for every pixel, but that would be much slower than working directly on the screen address to move up, down, left and right from a current pixel position.

Let's investigate further to substantiate that claim. Given a screen address in HL and a pixel mask in B, how would one move left one pixel? Here's the necessary code:

```
; hl = screen address, B = pixel mask
.left
   rlc b
   ret nc
   dec l
   ret
```

The pixel mask is rotated left one bit. This will be a valid pixel position unless B was already at the leftmost pixel position in the screen byte (ie B=1000 0000). The "RLC B" instruction will set the carry flag in that case and leave B=0000 0001. We use the no carry flag to return early if the new mask is valid, otherwise we update the column position one character to the left by decreasing the "CCCCC" portion of the screen address. The value of B at this point is 0000 0001, correctly masking the rightmost pixel in the new screen byte to the left of the old one. These four instructions are clearly quicker than rerunning the screen address subroutine. Notice that this subroutine doesn't check if it runs off the edge of the screen. The right pixel movement is similar, substituting "rrc b" for "rlc b" and "inc l" for "dec l".

To move up a pixel we need to decrement the Y coordinate, as pictured in figure 2. Given a screen address, this means first decreasing SSS followed by LLL (if necessary) and finally BB (if necessary). These bits are scattered about in the screen address pictured in figure 1 so a little care must be taken. The necessary code is shown here:

```
; hl = screen address

.SPPixelUp
   ld a,h          ; A=H=010BBSSS
   dec h           ; decrease SSS
   and $07         ; if SSS was not originally 000
   ret nz          ; we're done
   ld a,$08        ; otherwise SSS=111 (correct)
   add a,h         ; and we fix BB in H (one was subtracted)
   ld h,a
   ld a,l          ; A=X coord=LLLCCCCC
   sub $20         ; decrease LLL
   ld l,a
   ret nc          ; if no carry, LLL was not originally 000, okay
   ld a,h          ; otherwise LLL=111 now, that's okay
   sub $08         ; but need to decrease screen block
   ld h,a
   ret
```

This subroutine derives a lot of speed by minimizing the number of instructions executed in the most common cases. For example, 7 out of 8 times, only the first four instructions will be executed. 7 out 64 times, the first 11 instructions will

execute and the rest of the time (1 out of 64) all the instructions will execute. This makes the subroutine much quicker than one would initially guess by looking at the size of the code. The PixelDown subroutine is similar but is not shown here. All these pixel movement routines are reprinted in full in the floodfill listings elsewhere in this article.

That's enough information to have a first crack at a machine code version of the Basic flood fill routine.

Machine Code Flood Fills

Figure 3 contains a direct conversion of the Basic flood fill we saw earlier. No optimization has been done but it has been improved slightly to check for moving across screen boundaries. Type in the associated Basic listing to see it in action.

We have managed to speed things up considerably by moving to machine code, but there are still a couple of improvements that can be made. First, we compute screen addresses for every single point plotted. Since we always move up, down, left or right from the current pixel we could speed things up by avoiding this computation as discussed above.

The other optimization we can make is to plot 8 pixels at a time rather than one. Recall that each screen byte holds eight pixels. Why fiddle with it eight times to plot eight pixels in it when we could plot all 8 pixels at once with a single write of a whole byte?

The secret to plotting multiple pixels at once is the bytefill subroutine. It operates directly on a screen address and pixel mask, exactly what we will have available now that we have decided not to compute the screen address for every single pixel plotted.

```
; hl = screen address
;  b = incoming pixel mask

.Bytefill
   ld a,b          ; get pixel mask
   xor (hl)        ; zero out incoming pixels that
   and b           ; run into set pixels in display
   ret z           ; if no pixels left, ret

.bfloop            ; carry flag never set here
   ld b,a          ; b = incoming pixels
   rra             ; expand incoming pixels
   ld c,a          ; to the right and left
   ld a,b          ; within byte
   add a,a
   or c
   or b            ; a = incoming pixels wiggled
   ld c,a          ; save in c
   xor (hl)        ; zero out pixels that run into
```

```
and c           ; set pixels on display
cp b            ; have pixels changed from last loop?
jr nz, bfloop   ; keep going until incoming does not change

or (hl)
ld (hl),a       ; fill byte on screen
scf             ; indicate that this was a viable step
ret
```

Bytefill is called with a screen address in HL and a pixel mask containing all the "incoming" pixels. The incoming pixels are those pixels from where the flood fill grows in the current screen byte. Previously the flood fill always grew from a single point, but not anymore. The origin of the incoming byte will be clear while perusing the second flood fill listing in figure 4.

The Bytefill routine takes the incoming pixels and "wiggles" them to the left and right, trying to grow them into blank spaces within the screen byte. It does this until no more growth is possible within the screen byte. It then plots all those pixels and returns.

Putting these ideas into action produces figure 4, a byte-at-a-time flood fill routine. Type in the Basic listing to see it in action.

This routine is blazing fast; you will not see anything quicker. But, and this is a big but, there is a major flaw in the program that is shared with all the previous fills we have seen so far: the recursion depth is huge.

Consider a flood fill from the bottom left corner of a blank screen. According to figure 4, the first thing that is done is the fill of the entire screen byte in the bottom left corner. Then a move to the right is made and its byte is filled in a recursive call to "fill". Followed by another right move and call to "fill", then another, until we hit the right edge of the screen. A right move is not possible from the right edge of the screen so a left move is tried from there. That is unsuccessful because it was just filled. An up movement from the right edge is tried, successfully. Now we are at the right edge, one pixel up from the bottom of the screen. The filler fills in the byte and tries a right movement. That's not possible because a screen boundary is hit so it successfully tries a left movement. If this is carried on you'll notice that, from the bottom left corner of a blank screen, the screen is filled alternately from the left to the right and then from the right to the left as the fill line moves one pixel higher for each scan line filled.

You may have noticed that not a single return instruction is executed during the entire screen fill. That is a problem. Each call to fill puts at least 2 bytes on the stack to remember the return address. Since no returns are made for all screen bytes on the screen, there are 6144 calls made to fill without a single return. That's a recursion depth of 6144! Since at least 2 bytes are saved on the stack for each recursive call, at least 12288 bytes are needed to complete the fill. The

situation can be much worse, however. In the worst case movement (up or down) 6 bytes are saved on a recursive call (two for BC, two for HL and two for the return address). We may need up to 36864 bytes to fill the screen! The truth is somewhere in the middle. Notice that by moving from a pixel fill to a byte fill we have reduced the depth of recursion by a factor of eight since entire screen bytes are considered rather than individual pixels. Still this amount of memory usage is unacceptable for most applications. How can you use a flood fill in your own programs if it's going to need most of the available memory to complete?

This recursive fill is an example of a depth-first algorithm. It fills an area by going as deep as possible into the area (one call begets another call begets another, etc. without returning). The result is a memory requirement, computed as recursion depth times size of state information for each recursive step, that is proportional to the area to be filled. We can rescue the situation by considering another algorithmic approach, known as a breadth-first algorithm. Instead of going as deep as possible into an area, we try going wide first. This sounds like a lot of metaphysical talk of questionable value, but the terms "depth-first" and "breadth-first" are bonafide jargon that is used to describe the solution behaviour of many kinds of algorithms.

A breadth-first approach to a flood fill would try to fill all points in the immediate area first. From a starting point, all pixels to the immediate left, right, top and bottom are filled. Then for all those adjacent pixels, their immediate neighbours are filled, etc. The savings come from a key observation: once the immediate neighbours of a pixel are filled, there is no need to remember (come back to) the current pixel. Its information can be forgotten. This was not possible in the previous depth-first approaches. As will be seen later, the breadth-first approach will have a memory requirement proportional to the circumference of the area being filled, a significant savings.

The Breadth-First Approach

To implement the breadth-first approach, we will need to introduce a queue to hold future pixel positions that need investigating. From a currently filled screen byte (imagine the first screen byte to get the ball rolling), each direction should be investigated for possible flood fill expansion. If a move in any direction is possible (ie no pixel boundary was met), the surrounding pixel should be filled and added to the end of the queue for later investigation. Once all directions have been looked into, the next screen byte to investigate is retrieved from the front of the queue. Its immediate neighbours are then investigated with potental expansion pixels added to the end of the queue as before. This loop is repeated until there are no more screen bytes to investigate, indicated by an empty queue.

Figure 5 is the implementation. Run the Basic program to see it in action. We have lost a little speed, but you'll notice that the fill seems to progress in a saner manner. In the previous version the fill spread out all over the place. Now the fill

expands along a diamond-shaped boundary that grows away from the starting pixel. A little thought will reveal that, at any moment, the screen bytes in the queue are those screen bytes around the edge of the expanding diamond-shaped fill boundary. As a result, the necessary queue size to fill an area is roughly proportional to the circumference of the area to be filled.

The implementation in figure 5 allocates space on the stack for a queue whose size is determined by the caller. If at any moment the queue is found not to be large enough to complete the fill, the fill is aborted. This pleasant side effect of the breadth-first approach allows the caller to control how much memory is available for the fill. In previous versions there was no control, with the fill taking as much memory as needed.

With the necessary queue size to fill an area proportional to the circumference of the area to be filled, I have found the queue size can be as small as 100 screen positions for a complete fill of a typical screen. At three bytes per screen position, that adds up to a total memory requirement of about 300 bytes, a vast improvement over the previous requirement of between 12288 and 36864 bytes!

The Pattern Fill

This flood fill would become much more interesting if we were able to apply a pattern while filling an area rather than being stuck with the same old black. That would be the next logical step to take and really it's a small one compared to the large steps we have taken so far.

The first question to answer is how can we apply a pattern to the fill region? The procedure is fairly straightforward. Before writing the solid black fill byte to each screen position, logically AND it with a pattern byte. If the pattern is "10101010" and the fill byte is "00011111", the screen should be written with the logical AND of the two: "00001010". If the pattern were just a single byte, however, we could never make a filled area appear as anything more interesting than a collection of vertical stripes using this method.

To add some more variation I decided on an 8x8 pixel pattern defined in the same way as a UDG graphic. To determine which byte of the pattern UDG to use, the scan line bits in the screen address of the fill byte is used as an index. This is ideal since the scan line bits iterate through 0-7 repeatedly from the top to the bottom of the screen. For example, if the pattern UDG is stored beginning at address "x" and the screen address is held in register HL, then the pattern byte to use is stored at address "x + H&0x7" where "&" represents a logical AND. This byte should be read and ANDed with the fill byte, followed by a write to the screen to get the desired effect.

It is not quite as simple as that, however. With the breadth-first algorithm we have now, the flood fill expands from a diamond-shaped boundary centered on

the start pixel. If the naïve approach is taken, as suggested above, and the screen is written with a patterned fill byte, the boundary surrounding the start pixel would have holes in it wherever the pattern byte held a '0' bit. What would prevent the flood fill from expanding through those holes back to the start pixel from where it came? Nothing! The result could be a flood fill that constantly grew into and out of itself, possibly never terminating!

This problem occurs because of holes in the outermost boundary of the fill area. To avoid this problem, the outermost boundary needs to be kept black, with the pattern only applied to pixels in the interior of this boundary.

The implementation in figure 6 maintains three regions within the queue, called the new block, the investigate block and the pattern block. Each block is delimited by a special sentinel to indicate block boundaries within the queue. The investigate block contains the outermost boundary screen bytes from where the flood fill is growing. This corresponds to the set of screen bytes that were in the queue in the previous black flood-fill. As before, the flood fill attempts to expand from each screen byte in the investigate block in all directions; a successful expansion is added to the new block in the queue. The pattern block contains all those screen bytes that were previously investigated, representing the former outermost boundary. Initially the new block and the pattern block are empty and the investigate block contains the single screen byte representing the starting point of the fill.

Fill bytes in the new block are written as solid black to the screen. Once the investigate block has been completely investigated, the screen bytes in the pattern block (currently all black) have the pattern applied to them. Then the points in the investigate block become the pattern block and the new block becomes the investigate block. The loop repeats until the pattern block is empty. This algorithm maintains a solid black boundary two pixels thick around the fill region. The outermost boundary is the investigate block from where the fill grows and the innermost boundary is the pattern block, kept black to avoid an inward growth by the fill algorithm.

The queue now needs to be large enough to hold three circumferences of the fill region, one circumference per block in the queue. Previously it was found that 100 queue positions were needed per circumference so the pattern fill will need about 300 queue positions to complete most fills on screen. That corresponds to about 900 bytes of memory, still an acceptable memory requirement for almost all applications!

The assembly listing in figure 6 is rather large so I have not produced a Basic listing that you can type in and run to see the pattern fill in action. Instead I have supplied a C program that calls the pattern fill subroutine made available through the Sprite Pack library. The Sprite Pack library is a collection of various assembly language subroutines I have written over the years and made available

as a C library for C programs compiled using z88dk. Another article in this issue of ZQA explains this further. You can see the demo in action on your ts2068 or an emulator by downloading the compiled program from my website at "http://justme895.tripod.com/zqa/fillprogs.zip". This zip file contains "pfill.tap" which can be loaded and run in an emulator or on the real machine as described in the other article. A screenshot of the demo in action can be seen elsewhere in this article.

Figure 3. Pixel Coordinate Depth-First Flood Fill

```
; SPGetScrnAddr is not reprinted here to reduce the
; size of the article.  It can be found in the article text.

.test
   ld h,96              ; starting y coordinate
   ld l,128             ; starting x coordinate

; Flood Fill Version 1
; H = Y coord 0..191, L = X coord 0..255

.flood1
   push hl              ; save (x,y) coordinate
   ld a,h               ; GetScrnAddr requires A=H
   call SPGetScrnAddr   ; compute screen address
   pop hl               ; restore (x,y) in HL
   ld a,(de)            ; byte on screen
   and b                ; check if this pixel is set
   ret nz               ; if so, hit boundary so ret

   ld a,(de)            ; get screen byte
   or b                 ; set this pixel
   ld (de),a            ; plot it on screen

.right
   inc l                ; move pixel coord right
   call nz, flood1      ; if no wrap 255->0
   dec l                ; restore x coord
.left
   dec l                ; move pixel coord left
   ld a,l
   inc a
   call nz, flood1      ; if no wrap 0->255
   inc l                ; restore x coord
.up
   dec h                ; move pixel coord up
   ld a,h
   inc a
   call nz, flood1      ; if no wrap 0->255
   inc h                ; restore y coord
.down
   inc h                ; move pixel coord down
   ld a,h
   cp 192
   call c, flood1       ; if less than 192
   dec h                ; restore y coord
   ret


  10 REM COPY MACHINE CODE INTO MEMORY
  20 FOR n=32768 TO 32850: READ a: POKE n,a: NEXT n
  30 REM DRAW SOME CIRCLES ON DISPLAY
  40 FOR n=1 TO 10
  50 LET x=INT (RND*256)
  60 LET y=INT (RND*176)
```

```
  70 LET r=INT (RND*40)
  80 IF x-r<0 OR y-r<0 OR x+r>255 OR y+r>175 THEN
     GO TO 50
  90 CIRCLE x,y,r: NEXT n: CIRCLE 127,88,87
  95 REM THIS LAST CIRCLE IS NEEDED TO ENFORCE A MAXIMUM
     SIZE FILL REGION, OTHERWISE THERE MAY NOT BE ENOUGH
     MEMORY FOR THE FILL WHICH WOULD LEAD TO A CRASH
 100 LET a=USR 32768: PAUSE 50: RUN

1000 DATA 38,96,46,128,229,124,205,44,128,225,26
1010 DATA 160,192,26,176,18,44,196,4,128,45,45
1020 DATA 125,60,196,4,128,44,37,124,60,196,4,128
1030 DATA 36,36,124,254,192,220,4,128,37,201
1035 REM SPGetScrnAddr
1040 DATA 230,7,246,64,87,124,31,31,31,230,24,178
1050 DATA 87,125,230,7,71,62,128,40,3,31,16,253
1060 DATA 71,203,61,203,61,203,61,124,23,23,230
1070 DATA 224,181,95,201
```

Figure 4. Byte-At-A-Time Depth-First Flood Fill

```
; SPGetScrnAddr, Bytefill and SPPixelDown are not reprinted here
; to reduce the size of the article.  They can be found in the
; article text.

.test
   ld h,96              ; start pixel at centre of screen
   ld l,128

; byte at a time fill
; h = y coord, l = x coord

.flood2
   ld a,h
   call SPGetScrnAddr   ; b = pixel mask
   ex de,hl             ; hl = screen address

.fill
   call Bytefill        ; wiggle around incoming pixel mask
   ret nc               ; if incoming pixels hit boundary, ret

.up
   push hl              ; save screen address
   call SPPixelUp       ; move up one pixel
   jr c, offscreen1
   push bc              ; save pixel mask
   call fill            ; try to fill from new screen position
   pop bc               ; moving up, pixel mask remains same
.offscreen1
   pop hl

.down                   ; a replay of up
   push hl
   call SPPixelDown
   jr c, offscreen2
   push bc
   call fill
   pop bc
.offscreen2
   pop hl

.right
   bit 0,b              ; if first pixel in mask set, try right
   jr z, left
   inc l                ; move right one byte
   ld a,l               ; have we wrapped off screen?
   and $1f              ; (if so, CCCCC=0 now)
   jr z, offscreen3
   push bc              ; save current pixel mask
   ld b,$80             ; new incoming mask = leftmost pixel set
   call fill            ; fill from new screen position
   pop bc
.offscreen3
   dec l
```

```
.left                    ; a replay of right
   bit 7,b
   ret z
   ld a,l
   and $1f
   ret z
   dec l
   ld b,$01
   call fill
   inc l
   ret

; enter: HL = valid screen address
; exit : Carry = moved off screen
;        HL = moves one pixel up
; used : AF, HL

.SPPixelUp
   ld a,h
   dec h
   and $07
   ret nz
   ld a,$08
   add a,h
   ld h,a
   ld a,l
   sub $20
   ld l,a
   ret nc
   ld a,h
   sub $08
   ld h,a
   cp $40
   ret


  10 REM COPY MACHINE CODE INTO MEMORY
  20 FOR n=32768 TO 32941: READ a: POKE n,a: NEXT n
  30 REM DRAW SOME CIRCLES ON DISPLAY
  40 FOR n=1 TO 10
  50 LET x=INT (RND*256)
  60 LET y=INT (RND*176)
  70 LET r=INT (RND*40)
  80 IF x-r<0 OR y-r<0 OR x+r>255 OR y+r>175 THEN
     GO TO 50
  90 CIRCLE x,y,r: NEXT n: CIRCLE 127,88,87
 100 LET a=USR 32768: PAUSE 50: RUN

1000 DATA 38,96,46,128,124,205,70,128,235,205,109,128
1010 DATA 208,229,205,131,128,56,5,197,205,9,128,193
1020 DATA 225,229,205,152,128,56,5,197,205,9,128,193
1030 DATA 225,203,64,40,14,44,125,230,31,40,7,197,6
1040 DATA 128,205,9,128,193,45,203,120,200,125,230,31
1050 DATA 200,45,6,1,205,9,128,44,201
1055 REM SPGetScrnAddr
1060 DATA 230,7,246,64,87,124,31,31,31,230,24,178
1070 DATA 87,125,230,7,71,62,128,40,3,31,16,253
```

```
1080 DATA 71,203,61,203,61,203,61,124,23,23,230
1090 DATA 224,181,95,201
1095 REM ByteFill
1100 DATA 120,174,160,200,71,31,79,120,135,177
1110 DATA 176,79,174,161,184,194,113,128,182,119
1120 DATA 55,201
1125 REM SPPixelUp
1130 DATA 124,37,230,7,192,62,8,132,103,125,214
1140 DATA 32,111,208,124,214,8,103,254,64,201
1145 REM SPPixelDown
1150 DATA 36,124,230,7,192,124,214,8,103,125,198
1160 DATA 32,111,208,124,198,8,103,254,88,63,201
```

Figure 5. Breadth-First Black Flood Fill

```
; GetScrnAddr, Bytefill, PixelUp and PixelDown have
; been omitted for brevity.  You can find them
; elsewhere in this article.

.test
   ld l,128        ; x coord
   ld h,96         ; y coord
   ld bc,100       ; queue size
   call ffill
   ret

; enter: h = y coord, l = x coord, bc = queue size
; used : ix, af, bc, de, hl
; exit : this version does not bail, but portions of the screen may not
be
;        filled if the queue size was too small
; stack: 3*bc+12 bytes, not including the call to ffill or interrupts

.ffill
   ld a,h
   cp 192
   ret nc           ; if y coord out of bounds
   dec bc           ; we will start with one struct in the queue
   push bc          ; save max stack depth variable
   call getscrnaddr ; de = screen address, b = pixel byte
   ex de,hl         ; hl = screen address
   call bytefill    ; b = fill byte
   jr c, viable
   pop bc
   ret

.viable
   ld ix,-1
   add ix,sp        ; ix = top of queue = initial investigate block
   push hl          ; screen address and fill byte are
   push bc          ;   first struct in investigate block
   inc sp
   xor a
   push af          ; mark end of investigate block
   dec sp

   ld c,(ix+1)      ; reserve space on stack for queue
   ld b,(ix+2)      ; bc = max stack depth - 1
   inc bc
   ld l,c
   ld h,b
   add hl,bc        ; space required = 3*BC (max depth) + 7
   add hl,bc        ;   but have already taken 6 bytes and the
   ld c,l           ;   queue end marker is pushed below.
   ld b,h           ; bc = # uninitialized bytes in queue
   ld h,a
   ld l,a           ; hl = 0
   sbc hl,bc        ; hl = -bc
   add hl,sp
```

```
   ld (hl),a        ; zero last byte in queue
   ld sp,hl         ; move stack below queue
   ld a,$80
   push af          ; mark end of queue with $80 byte
   inc sp
   ld e,l
   ld d,h
   inc de
   dec bc
   ldir             ; zero the uninitialized bytes in queue

; NOTE: Must move the stack before clearing the queue, otherwise an
interrupt
; may overwrite portions of the cleared queue.

   push ix
   pop bc           ; bc = top of queue
   ld hl,-6
   add hl,bc
   ex de,hl         ; de = new block
   ld l,c
   ld h,b           ; hl = investigate block

; ix = top of queue, bottom of queue marked with $80 byte
; hl = investigate block, de = new block

; Variables indexed by ix, LSB first:
;   ix + 03/04    return address
;   ix + 01/02    max stack depth

; A picture of memory at this point:
;
;+-----------------------+   higher addresses
;|                       |         |
;|-   return address    -|        \|/
;|                       |         V
;+-----------------------+   lower addresses
;|                       |
;|-  max stack depth    -|
;|                       |
;+-----------------------+
;|  screen address MSB   |  <- ix = top of queue = hl = investigate
block
;|  screen address LSB   |
;|      fill byte        |
;+-----------------------+
;|  end of block marker  |
;|          ?            |
;|          ?            |
;+-----------------------+
;|          0            |  <- de = new block
;|          0            |
;|          0            |
;+-----------------------+
;|                       |
;|        ......         |  size is a multiple of 3 bytes
;|     rest of queue     |
```

```
;|      all zeroed       |
;|        ......         |
;|                       |
;+-----------------------+
;|         $80           |  <- sp, special byte marks end of queue
;+-----------------------+


.pfloop
   ld a,(hl)
   cp $80          ; bit 15 of screen addr set if time to wrap
   jr c, inowrap
   push ix
   pop hl          ; hl = ix = top of queue
   ld a,(hl)
.inowrap
   cp $40          ; screen address < $4000 marks end of block
   jr c, endinv    ; are we done yet?
   ld b,a
   dec hl
   ld c,(hl)       ; bc = screen address
   dec hl
   ld a,(hl)       ; a = fill byte
   dec hl
   inc (ix+1)      ; increase available queue space by one
   jr nz, bcnowrap
   inc (ix+2)
.bcnowrap
   push hl         ; save spot in investigate block
   ld l,c
   ld h,b          ; hl = screen address
   ld b,a          ; b = fill byte

.goup
   push hl         ; save screen address
   call pixelup    ; move screen address up one pixel
   jr c, updeadend ; if went off-screen
   push bc         ; save fill byte
   call bytefill
   call c, addnew  ; if up is not dead end, add this to new block
   pop bc          ; restore fill byte
.updeadend
   pop hl          ; restore screen address

.godown
   push hl         ; save screen address
   call pixeldown  ; move screen address down one pixel
   jr c, downdeadend ; if went off-screen
   push bc         ; save fill byte
   call bytefill
   call c, addnew   ; if down is not dead end, add this to new block
   pop bc           ; restore fill byte
.downdeadend
   pop hl           ; restore screen address

.goleft
```

```
   bit 7,b          ; can only move left if leftmost bit of fill byte
set
   jr z, goright
   push hl          ; save screen address
   ld a,l
   dec l            ; decrease column
   and 31
   jr z, leftdeadend ; if went off-screen
   push bc          ; save fill byte
   ld b,$01         ; set rightmost pixel for incoming byte
   call bytefill
   call c, addnew   ; if left is not dead end, add this to new block
   pop bc           ; restore fill byte
.leftdeadend
   pop hl           ; restore screen address

.goright
   bit 0,b          ; can only move right if rightmost bit of fill byte
set
   jr z, nextinv
   inc l            ; next column
   ld a,l
   and 31
   jr z, nextinv    ; if went off-screen
   ld b,$80         ; set leftmost pixel for incoming byte
   call bytefill
   call c, addnew   ; if right is not dead end, add this to new block

.nextinv
   pop hl           ; hl = spot in investigate block
   jr pfloop

.endinv
   dec hl
   dec hl
   dec hl           ; investigate block now points at new block

   ld a,(de)        ; check if new block is at end of queue
   cp $80
   jr c, nowrapnew
   push ix
   pop de           ; de = ix = top of queue
.nowrapnew
   xor a
   ld (de),a        ; store end marker for new block
   dec de
   dec de
   dec de

   ld a,(hl)        ; done if the investigate block is empty
   cp $40
   jr nc, pfloop

.endpfill
   ld sp,ix
   inc sp
   inc sp
```

```
   inc sp           ; return address at ix+3
   ret


; add incoming fill byte and screen address to new block
; enter b = incoming byte, hl = screen address, de = new block

.addnew
   push hl          ; save screen address
   ld l,(ix+1)
   ld h,(ix+2)      ; hl = max stack depth
   ld a,h
   or l
   jr nz, stillroom ; this version doesn't bail
   pop hl           ; just don't add to new block
   ret

.stillroom
   dec hl           ; available queue space decreases by one struct
   ld (ix+1),l
   ld (ix+2),h
   pop hl           ; hl = screen address

   ld a,(de)        ; check if new block is at end of queue
   cp $80
   jr c, annowrap
   push ix
   pop de           ; de = ix = top of queue
.annowrap
   ex de,hl
   ld (hl),d        ; make struct, store screen address (2 bytes)
   dec hl
   ld (hl),e
   dec hl
   ld (hl),b        ; store fill byte (1 byte)
   dec hl
   ex de,hl
   ret


10 REM COPY MACHINE CODE INTO MEMORY
  20 FOR n=32768 TO 33121: READ a: POKE n,a: NEXT n
  30 REM DRAW SOME CIRCLES ON DISPLAY
  40 FOR n=1 TO 10
  50 LET x=INT (RND*256)
  60 LET y=INT (RND*176)
  70 LET r=INT (RND*40)
  80 IF x-r<0 OR y-r<0 OR x+r>255 OR y+r>175 THEN
     GO TO 50
  90 CIRCLE x,y,r: NEXT n: CIRCLE 127,88,87
 100 LET a=USR 32768: PAUSE 50: RUN

 995 REM TEST
1000 DATA 46,128,38,96,1,100,0,205,11,128,201
1005 REM FFILL
1010 DATA 124,254,192,208,11,197,205,249,128,235
1020 DATA 205,75,129,56,2,193,201,221,33,255,255
```

```
1030 DATA 221,57,229,197,51,175,245,59,221,78,1
1040 DATA 221,70,2,3,105,96,9,9,77,68,103,111,237
1050 DATA 66,57,119,249,62,128,245,51,93,84,19,11
1060 DATA 237,176,221,229,193,33,250,255,9,235,105
1070 DATA 96,126,254,128,56,4,221,229,225,126,254,64
1080 DATA 56,91,71,43,78,43,126,43,221,52,1,32,3,221
1090 DATA 52,2,229,105,96,71,229,205,32,129,56,8,197
1100 DATA 205,75,129,220,211,128,193,225,229,205
1110 DATA 53,129,56,8,197,205,75,129,220,211,128,193
1120 DATA 225,203,120,40,18,229,125,45,230,31,40,10
1130 DATA 197,6,1,205,75,129,220,211,128,193,225,203
1140 DATA 64,40,14,44,125,230,31,40,8,6,128,205
1150 DATA 75,129,220,211,128,225,24,152,43,43,43,26
1160 DATA 254,128,56,3,221,229,209,175,18,27,27,27
1170 DATA 126,254,64,48,131,221,249,51,51,51,201
1175 REM ADDNEW
1180 DATA 229,221,110,1,221,102,2,124,181,32,2,225,201
1190 DATA 43,221,117,1,221,116,2,225,26,254,128,56,3
1200 DATA 221,229,209,235,114,43,115,43,112,43,235,201
1205 REM GETSCRNADDR
1210 DATA 230,7,246,64,87,124,31,31,31,230,24,178
1220 DATA 87,125,230,7,71,62,128,40,3,31,16,253
1230 DATA 71,203,61,203,61,203,61,124,23,23,230
1240 DATA 224,181,95,201
1245 REM PIXELUP
1250 DATA 124,37,230,7,192,62,8,132,103,125,214
1260 DATA 32,111,208,124,214,8,103,254,64,201
1265 REM PIXELDOWN
1270 DATA 36,124,230,7,192,124,214,8,103,125,198
1280 DATA 32,111,208,124,198,8,103,254,88,63,201
1285 REM BYTEFILL
1290 DATA 120,174,160,200,71,31,79,120,135,177
1300 DATA 176,79,174,161,184,71,194,80,129,182,119
1310 DATA 55,201
```

Figure 6. Breadth-First Pattern Flood Fill

```
; In the interest of brevity, the source for getscrnaddr, pixelup,
; pixeldown and bytefill is not reprinted here.

; Each entry in the queue is a 3-byte struct that grows down in memory:
;       screen address      (2-bytes, MSB first)
;       fill byte           (1-byte)
; Screen address with MSB<0x40 is used to indicate the end of a block.
; Screen address with MSB>=0x80 is used to mark the physical end of Q.
;
; The fill pattern is a typical 8x8 pixel character, stored in 8 bytes.

; enter: h = y coord, l = x coord, bc = queue size, de = address of
fill pattern
;        In hi-res mode, carry flag is most significant bit of x coord
; used : ix, af, bc, de, hl
; exit : no carry = success, carry = had to bail queue was too small
; stack: 3*bc+30 bytes, not including the call to PFILL or interrupts

.SPPFill
   push de           ; save (pattern pointer) variable
   dec bc            ; we will start with one struct in the queue
   push bc           ; save max stack depth variable

   ld a,h
   call SPGetScrnAddr ; de = screen address, b = pixel byte
   ex de,hl          ; hl = screen address
   call bytefill     ; b = fill byte
   jr c, viable
   pop bc
   pop de
   ret

.viable
   ex de,hl          ; de = screen address, b = fill byte
   ld hl,-7
   add hl,sp
   push hl           ; create pattern block pointer = top of queue
   push hl
   pop ix            ; ix = top of queue
   dec hl
   dec hl
   dec hl
   push hl           ; create investigate block pointer
   ld hl,-12
   add hl,sp
   push hl           ; create new block pointer

   xor a
   push af
   dec sp            ; mark end of pattern block
   push de           ; screen address and fill byte are
   push bc           ;   first struct in investigate block
   inc sp
   push af
```

```
   dec sp            ; mark end of investigate block

   ld c,(ix+7)
   ld b,(ix+8)        ; bc = max stack depth - 1
   inc bc
   ld l,c
   ld h,b
   add hl,bc          ; space required = 3*BC (max depth) + 10
   add hl,bc          ; but have already taken 9 bytes
   ld c,l
   ld b,h             ; bc = # uninitialized bytes in queue
   ld hl,0
   sbc hl,bc          ; negate hl, additions above will not set carry
   add hl,sp
   ld (hl),0          ; zero last byte in queue
   ld sp,hl           ; move stack below queue
   ld a,$80
   push af            ; mark end of queue with $80 byte
   inc sp
   ld e,l
   ld d,h
   inc de
   dec bc
   ldir               ; zero the uninitialized bytes in queue

; NOTE: Must move the stack before clearing the queue, otherwise an
interrupt could overwrite portions of the (just cleared) queue.

; ix = top of queue, bottom of queue marked with 0x80 byte

; Variables indexed by ix, LSB first:
;   ix + 11/12    return address
;   ix + 09/10    fill pattern pointer
;   ix + 07/08    max stack depth
;   ix + 05/06    pattern block pointer
;   ix + 03/04    investigate block pointer
;   ix + 01/02    new block pointer

; A picture of memory at this point:
;
;+-----------------------+   higher addresses
;|                       |        |
;|-   return address    -|       \|/
;|                       |        V
;+-----------------------+   lower addresses
;|        fill           |
;|-  pattern pointer    -|
;|                       |
;+-----------------------+
;|                       |
;|-  max stack depth    -|
;|                       |
;+-----------------------+
;|                       |
;|-   pattern block     -|
;|                       |
;+-----------------------+
```

```
;|                        |
;|- investigate block   -|
;|                        |
;+------------------------+
;|                        |
;|-     new block       -|
;|                        |
;+------------------------+
;|  end of block marker   |  <- ix = pattern block = top of queue
;|          ?             |
;|          ?             |
;+------------------------+
;|  screen address MSB    |  <- investigate block
;|  screen address LSB    |
;|      fill byte         |
;+------------------------+
;|  end of block marker   |
;|          ?             |
;|          ?             |
;+------------------------+
;|          0             |  <- new block
;|          0             |
;|          0             |
;+------------------------+
;|                        |
;|        ......          |  size is a multiple of 3 bytes
;|     rest of queue      |
;|      all zeroed        |
;|        ......          |
;|                        |
;+------------------------+
;|         0x80           |  <- sp, special byte marks end of queue
;+------------------------+

.pfloop
   ld l,(ix+3)
   ld h,(ix+4)       ; hl = investigate block
   ld e,(ix+1)
   ld d,(ix+2)       ; de = new block
   call investigate
   ld (ix+1),e
   ld (ix+2),d       ; save new block
   ld (ix+3),l
   ld (ix+4),h       ; save investigate block

   ld l,(ix+5)
   ld h,(ix+6)       ; hl = pattern block
   ld c,(ix+7)
   ld b,(ix+8)       ; bc = max stack depth (available space)
   call applypattern
   ld (ix+7),c
   ld (ix+8),b       ; save stack depth
   ld (ix+5),l
   ld (ix+6),h       ; save pattern block

   ld a,(hl)         ; done if the investigate block was empty
   cp 0x40
```

```
   jp nc, pfloop

.endpfill
   ld de,11           ; return address is at ix+11
   add ix,de
   ld sp,ix
   or a               ; make sure carry is clear, indicating success
   ret

; IN/OUT: hl = investigate block, de = new block

.investigate
   ld a,(hl)
   cp 0x80            ; bit 15 of screen addr set if time to wrap
   jp c, inowrap
   push ix
   pop hl             ; hl = ix = top of queue
   ld a,(hl)

.inowrap
   cp 0x40            ; screen address < 0x4000 marks end of block
   jp c, endinv       ; are we done yet?
   ld b,a
   dec hl
   ld c,(hl)          ; bc = screen address
   dec hl
   ld a,(hl)          ; a = fill byte
   dec hl
   push hl            ; save spot in investigate block
   ld l,c
   ld h,b             ; hl = screen address
   ld b,a             ; b = fill byte

.goup
   push hl            ; save screen address
   call SPPixelUp     ; move screen address up one pixel
   jr c, updeadend    ; if went off-screen
   push bc            ; save fill byte
   call bytefill
   call c, addnew     ; if up is not dead end, add this to new block
   pop bc             ; restore fill byte

.updeadend
   pop hl             ; restore screen address

.godown
   push hl            ; save screen address
   call SPPixelDown   ; move screen address down one pixel
   jr c, downdeadend
   push bc            ; save fill byte
   call bytefill
   call c, addnew     ; if down is not dead end, add this to new block
   pop bc             ; restore fill byte

.downdeadend
   pop hl             ; restore screen address
```

```
.goleft
   bit 7,b            ; can only move left if leftmost bit of fill byte
set
   jr z, goright
   ld a,l
   and 31
   jr nz, okleft
   bit 5,h            ; for hi-res mode: column = 1 if l=0 and bit 5 of
h is set
   jr z, goright

.okleft
   push hl            ; save screen address
   call SPCharLeft
   push bc            ; save fill byte
   ld b,0x01          ; set rightmost pixel for incoming byte
   call bytefill
   call c, addnew     ; if left is not dead end, add this to new block
   pop bc             ; restore fill byte
   pop hl             ; restore screen address

.goright
   bit 0,b            ; can only move right if rightmost bit of fill
byte set
   jr z, nextinv
   or a               ; clear carry
   call SPCharRight
   jr c, nextinv      ; went off screen
   ld a,l
   and 31
   jr z, nextinv      ; wrapped around line
   ld b,0x80          ; set leftmost pixel for incoming byte
   call bytefill
   call c, addnew     ; if right is not dead end, add this to new block

.nextinv
   pop hl             ; hl = spot in investigate block
   jp investigate

.endinv
   dec hl
   dec hl
   dec hl             ; investigate block now points at new block

   ld a,(de)          ; check if new block is at end of queue
   cp 0x80
   jr c, nowrapnew
   defb 0xdd
   ld e,l
   defb 0xdd
   ld d,h             ; de = ix = top of queue

.nowrapnew
   xor a
   ld (de),a          ; store end marker for new block
   dec de
   dec de
```

```
   dec de
   ret

; add incoming fill byte and screen address to new block
; enter b = incoming byte, hl = screen address, de = new block

.addnew
   push hl           ; save screen address
   ld l,(ix+7)
   ld h,(ix+8)       ; hl = max stack depth
   ld a,h
   or l
   jr z, bail        ; no space in queue so bail!
   dec hl            ; available queue space decreases by one struct
   ld (ix+7),l
   ld (ix+8),h
   pop hl            ; hl = screen address

   ld a,(de)         ; check if new block is at end of queue
   cp 0x80
   jr c, annowrap
   defb 0xdd
   ld e,l
   defb 0xdd
   ld d,h            ; de = ix = top of queue

.annowrap
   ex de,hl
   ld (hl),d         ; make struct, store screen address (2 bytes)
   dec hl
   ld (hl),e
   dec hl
   ld (hl),b         ; store fill byte (1 byte)
   dec hl
   ex de,hl
   ret

; if the queue filled up, we need to bail.  Bailing means patterning
any set pixels
; which may still be on the display.  If we didn't bail
; there is no guarantee the fill would ever return.

.bail
   pop hl            ; hl = screen address, b = fill byte
   ld a,b
   xor (hl)
   ld (hl),a         ; clear this byte on screen

   xor a
   ld (de),a         ; mark end of new block

   ld l,(ix+5)
   ld h,(ix+6)       ; hl = pattern block
   call applypattern ; for pattern block
   call applypattern ; for investigate block
   call applypattern ; for new block
```

```
   ld de,11          ; return address is at ix+11
   add ix,de
   ld sp,ix
   scf               ; indicate we had to bail
   ret

; hl = pattern block, bc = max stack depth (available space)

.applypattern
   ld a,(hl)
   cp 0x80           ; bit 15 of screen addr set if time to wrap
   jp c, apnowrap
   push ix
   pop hl            ; hl = ix = top of queue
   ld a,(hl)

.apnowrap
   cp 0x40           ; screen address < 0x4000 marks end of block
   jr c, endapply    ; are we done yet?

   and 0x07          ; use scan line 0..7 to index pattern
   add a,(ix+9)
   ld e,a
   ld a,0
   adc a,(ix+10)
   ld d,a            ; de points into fill pattern
   ld a,(de)         ; a = pattern

   ld d,(hl)
   dec hl
   ld e,(hl)         ; de = screen address
   dec hl

   and (hl)           ; and pattern with fill byte
   sub (hl)           ; or in complement of fill byte
   dec a
   ex de,hl
   and (hl)           ; apply pattern to screen
   ld (hl),a
   ex de,hl
   dec hl
   inc bc             ; increase available queue space
   jp applypattern

.endapply
   dec hl
   dec hl
   dec hl             ; pattern block now pts at investigate block
   ret


/* Pattern Fill Demo Program */
/* Alvin Albrecht 01.2003    */

/* C Program for ts2068 or Spectrum */
/* Compile with the z88dk compiler  */
/* http://z88dk.sourceforge.net/    */
```

```
#include <stdlib.h>
#include <graphics.h>
#include <spritepack.h>

/* first define some pattern UDGs */
extern uchar patterns[];
#asm
._patterns

defb @11111111
defb @11111111
defb @11111111
defb @11111111
defb @11111111
defb @11111111
defb @11111111
defb @11111111

defb @10101010
defb @01010101
defb @10101010
defb @01010101
defb @10101010
defb @01010101
defb @10101010
defb @01010101

defb @00000000
defb @01111110
defb @01100110
defb @01100110
defb @01100110
defb @01100110
defb @01111110
defb @00000000

defb @10001000
defb @01000100
defb @00100010
defb @00010001
defb @10001000
defb @01000100
defb @00100010
defb @00010001

defb @00010001
defb @00100010
defb @01000100
defb @10001000
defb @00010001
defb @00100010
defb @01000100
defb @10001000

defb @10011001
defb @01100110
```

```
defb @01100110
defb @10011001
defb @10011001
defb @01100110
defb @01100110
defb @10011001

defb @00100010
defb @01010101
defb @10001000
defb @00000000
defb @00100010
defb @01010101
defb @10001000
defb @00000000

defb @11111111
defb @10000000
defb @10100010
defb @10010100
defb @10001000
defb @10010100
defb @10100010
defb @10000000
#endasm


main()                          /* C programs start here */
{
   int x,y,r,n;

   while (1) {                  /* forever */
      clg();                    /* clear screen */
      for (n=0; n!=10; n++) {   /* for loop executed 10 times */
         do {
            x = rand() % 256;   /* pick centre coordinate of circle */
            y = rand() % 192;
            r = rand() % 40;    /* pick random radius */
         } while (((x-r)<0) || ((y-r)<0) || ((x+r)>255) || ((y+r)>191));
         circle(x,y,r,1);       /* draw random circle */
      }
      x = rand() % 256;
      y = rand() % 192;
      r = (rand() % 8)*8;       /* 1 of 8 patterns defined above */
      sp_PFill(x, y, patterns + r, 300);  /* pattern fill @ x,y */
      sp_WaitForKey();
   }
}
```