DEFENDER - 2018 BASIC 10-Liners Game Contest Entry

My first (and maybe only) entry to the 2018 10-Liner BASIC Game competition is an Atari Turbo-BASIC XL implementation of the 1981 arcade game DEFENDER.

It is entered into the EXTREM-256 line-length category. Defender is a side-scroller, which is why I chose to try it. I had been trying to figure out how to do fast horizontal scrolling with BASIC and eventually landed on the technique used here. Find it over on GitHub or download the auto-loading ATR file.

The Atari ANTIC video processor is driven by a micro program called a Display List (DL). The DL has one entry per line of graphics and defines a number of things including the graphics mode and an optional starting address for the data. As described in De Re Atari, one uses an expanded screen memory and merely specifies the starting address for each line of graphics to be displayed. In an assembly language program, the starting addresses can be updated during a vertical blank. The trouble with BASIC is that computing and poking addresses inside FOR-NEXT loop is not nearly fast enough to be satisfactory. I realized instead, I could build a precomputed look up table (LUT) and store it in RAM. Then using Turbo-BASIC's MOVE command, I could rapidly update the display list with the appropriate data to get fast side scrolling.

Once I had the side scrolling, everything else slowly fell into place. I created a random mountain generator. I wanted a radar display. Then I had to compromise on the AI's because the computational load was getting a little high for the game loop. I developed the AI's in graphics mode 0 with only 40 columns of playfield. This was enough to see the overall behavior and get a satisfactory result. Here's a screenshot of the final prototype. See the "(" kidnapping the "$"?

At this point, my program was 12 lines long. I settled on static bad guys except for special abductors who descend to kidnap your citizens. If you successfully terminate an abductor in the act, the citizen gracefully floats to earth. Be careful not to takeout a citizen accidentally - friendly fire is gruesome but challenging.

The Code

Here's the obfuscated 10-liner produced by DSMC's tbxl-parser.

The most straightforward code to analyze is here. It doesn't use any compression techniques and the AI code is separated out for each character type. This code parses into too many lines. The first task I did was to copy the compression techniques developed by Victor Parada for Space Ranger. This got be down another line. Then I tackled the AI logic, which had lots of duplicated commands to update the playfield and radar display. I matched up the AI's with the most common logic and screen update code and combined them together. This got me down to 10 lines plus the ?"game over" statement. I then found two POKES which I combined into a single DPOKE. That did it.

Here's the final source file with expanded comments:

```
'------------------------------------------------------------
'
'                          DEFENDER
'
'For Atari 800XL 8-Bit BASIC 10-Liner
'One-player game with joystick
'
'Jeff Piepmeier
'January 31, 2018
'http://jeffpiepmeier.blogspot.com/
'http://github.com/jeffpiep/
'
'--Parsed with TurboBASIC XL Parser Tool
'http://github.com/dmsc/tbxl-parser
$options +optimize
'-- Tested on Altirra
'http://www.virtualdub.org/altirra.html
'-- bitmap graphics developed with Mad Studio
'http://atariage.com/forums/topic/258728-mad-studio/
'-- POKE/MOVE compression technique copied/adapted
'from Victor Parada's Space Ranger
'http://www.vitoco.cl/atari/10liner/RANGER/
'------------------------------------------------------------
```

First is a very long string of ATASCII for storing addresses and data. The first 6 characters are a very short display list described below. Then there are 8 pokes and some player-missile data followed by some POKEY sound settings.

```
'a whole bunch of data for POKES and MOVES
A=ADR("\46\62\BF\41\50\A0\6F\A0\00\07\D4\B0\C0\02\08\1D\D0\03\6F\0
2\31\2F\02\3E\F4\02\B8\62\BF\E2\04\08\D0\01\03\03\03\04\C4\02\0E\3
6\D6\88\03\AC\A0\41\50\A0\04\56\A0\00\4E\50\A1\08\20\B8\30\38\30\2
0\70\20\20\70\08\28\B8\7C\D6\D6\7C\38\7C\54\92\14\2A\B3\90\90\90\9
0\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\06\04\B2\20\70\F
```

```
8\7E\FF\70\06\0E\B2\04\0E\1F\7E\FF\0E\04\00\D2\C0\AF\C1\AF\00")

'make a string of spaces to use a couple different places
DIM A$(4)
A$="    "
```

I use graphics mode 15 because it's well suited to the radar display and it zeros out a whole bunch of RAM. There's enough RAM to contain the radar, playfield, PM graphics and custom character set. Here's the memory map I made to keep track of everything:

C000----------
Text Screen
BF60----------
BC00----------
Character Set
B800----------
Player 3
B700----------
Player 2
B600----------
Player 1
B500----------
Player 0
B400----------
Missiles
B300----------
B140----------
Character GFX
A4C0----------
Bitmap GFX
A150----------
A0AB----------
Display List
A050----------
9FD0----------
Look Up Table
7F00----------

```
'base the memory map on graphics mode 15
GR.15
```

I move the text window to the first line of the screen and use graphics mode 2. The LUT takes so long to build, I decide to speed it up some by temporarily using a wait for vertical scan command right after the first line of text. This probably speeds things up 30-40%. I wanted to have a countdown displayed so the user would know the program was working.

```
'START: build new display list
'SHORTEN DISPLAY LIST SO STEAL FEWER CPU CYCLES one line of
graphics 1 mapped to text window; wait for vertical scan
'MOVE ADR("\46\62\BF\41\50\A0"),$A053,6
MOVE A,$A053,6
A=A+6
```

The side-scrolling feature needs a look-up table (LUT) whose entries can be quickly copied into the display list to create the scroll. This nest loop computes the LUT and stores in RAM below the video.

```
'write coarse-scrolling LUT to RAM below video
FIELDWIDTH = 160
LUT=$7F00
ADDR=LUT
LINEONE=42176 : REM start of character graphics in video ram
FOR COLS=0 TO 139
  ? 139-COLS;A$;
  FOR ROWS=0 TO 19
    POKE ADDR,86 : REM GR.1 CHARACTER MODE; DP. TAKES 1 LESS CHAR
THAN POKE
    DPOKE ADDR+1,LINEONE+COLS+ROWS*FIELDWIDTH
    ADDR=ADDR+3
  NEXT ROWS
  POKE 657,2
NEXT COLS
```

There are several POKE commands needed to set up the PM graphics - the addresses and data are stored in the giant string in the first line.

```
REM COMPRESSED CODE GOES HERE
FOR I = 1 TO 8
  POKE DPEEK(A),PEEK(A+2)
  A=A+3
NEXT I
```

The ROM character set is copied into RAM so it can be modified. The modifications are done by copying custom characters stored above into RAM and by remapping some of the characters to different locations. The remapping is done below.

```
'copy the ROM character set for modification
MOVE $E000,$B800,512
```

Copy data out of the string above in to RAM. This code is "borrowed" from http://www.vitoco.cl/atari/10liner/RANGER/. The first byte in the record is the number of bytes to be copied. The second two are the target 16-bit address. The remainder are the data. The process is repeated until a 0 is encountered. The very last move turns on two sound channels to create a beating low-E sound similar to the start of the arcade Defender.

```
REM # BYTES, TARGET ADDRESS (LO/HI), DATA
B=PEEK(A)
WHILE B
 MOVE A+3,DPEEK(A+1),B
 A=A+B+3
 B=PEEK(A)
WEND
```

I adapted this technique to be able to copy memory from one location to

another. I move the ATASCII line art characters to a different location in the character set and I copy the bad guy character to multiple locations in the map. I need multiple copies because I use the character code to store the AI identities in the screen memory itself. This avoids needed to keep track of too many things with arrays.

```
'copy some memory around, including remapping some characters
A=ADR("\30\E2\08\B8\10\68\E2\18\B8\08\28\B8\30\B8\18\00\B8\48\B8\0
8\03\B2\10\B9\08\62\BF\63\BF\02")
FOR I=1 TO 6
  MOVE DPEEK(A),DPEEK(A+2),PEEK(A+4)
  A=A+5
NEXT I
REM END COMPRESSED CODE
```

I wanted to have the mountains in the play field and though having the randomly generated might be more compact that storing a predetermined set. This loop randomly picks up/straight/down directions on each iteration and draws the mountains on the playfield and radar. The ATASCII line art characters were remapped so the screen codes are 1, 2 and 3. This makes poking them into RAM straightforward. Also, they needed to be moved into the lower 64 characters to be accessible in graphics mode 2. Finally, note here and everywhere else below there are both POKE and PLOT commands to put things on the playfield and radar display. This dual display code in part forces me into the EXTREM-256 category.

```
'draw the mountains
QUARTER = 39 : REM 79-FIELDWIDTH/4, used in the radar display
COLOR 2
Y=1
FOR I=0 TO (FIELDWIDTH-1)
  'choose random slope
  R=RAND(3)+1
  'check for top or bottom boundaries
  R=R+(Y=0)*(R=1)-(Y=5)*(R=2)
  'increase height if needed
  Y=Y-(R=1)
  'insert character into last 5 rows of map
  POKE $A4C0+(Y+14)*FIELDWIDTH+I,R+64
  PLOT QUARTER+I/2,15+Y
  'decrease height if needed
  Y=Y+(R=2)
NEXT I
```

This little bit draws the outline for the radar display. The radar display was an important feature to have for me. Besides just being darn cool, it allows the player to see if his citizens are being kidnapped.

```
'draw the RADAR outline
COLOR 3
PLOT 0,21 : DRAWTO 159,21
```

```
DRAWTO 78-40,21 : DRAWTO 78-40,0
DRAWTO 80+40,0 : DRAWTO 80+40,21
```

To speed up AI processing, I store abductors and citizens in a 2-D array. The abductors are initially hidden and coded with a value of 9 in the screen memory. The players are placed on the screen above the mountains so they can drop down. I take advantage of the AI behavior so I don't have to remember where the mountains are.

```
'generate abductors and victims
NA = 9
DIM AI(NA,2)
FOR I=0 TO NA-1
  REPEAT
    PX = RAND(FIELDWIDTH) : REM X LOCATION OF PEOPLE
    PPX = LINEONE+13*FIELDWIDTH+PX
  UNTIL PEEK(PPX)=0
  AI(I,1) = PPX : REM PLAYFIELD ADDRESS
  POKE PPX,$04 : REM INSERT PEOPLE
  AI(I,2) = LINEONE+PX
  POKE LINEONE+PX,NA : REM INSERT LATENT ABDUCTOR, NA HAPPENS TO =
9 which is char code
NEXT I
```

I initially tried to have the other bad guys move around, but it just was too slow, took too much code space and didn't add much to the game play. So they are static and simply inserted into the screen. There's a small chance of 1:1440 that one will be placed right where the hero's ship starts. It's only happened to me once during testing. I didn't try to squeeze in logic to avoid that - I suppose a single POKE of a 0 into that location would do the job. Alas.

```
'generate some static bad guys
'always slim chance one lands on ship at start, but that's a
feature
FOR I=1 TO 20
  REPEAT
    X = 10+RAND(FIELDWIDTH-20)
  UNTIL PEEK(LINEONE+X)=0
  Y = 1+RAND(12)
  BB = LINEONE+X+Y*FIELDWIDTH
  POKE BB, $85 : REM PUT DOWN BADDIES
  PLOT QUARTER+((BB-LINEONE) MOD FIELDWIDTH)/2,Y+1
NEXT I
```

This bit initializes the game play: sets the starting location and direction of the ship; initialize the citizen and lives counters (the score starts at 0 already since you don't have to initialize variables in BASIC). The countdown timer is set to 5000.

```
'start the ship in the middle of the play field
XWLD = 70
'point the ship to the left
DX=-1
```

```
PY=$3E : REM POSITION THE Y-LOCATION SO SHIP LOCATIONS OVERLAP
CHARACTER LOCATIONS
'SCORE=0
PEOPLE=NA
LIVES=3
POKE $D000,120
PM=$B440:REM PM*$100+$200 TO POINT TO FIRST PLAYER
T0=TIME+5E3
```

And turn off the sound before play starts. I had so much code below, I didn't even try to create sound effects.

```
'turn off the sound
SOUND
```

Whew, finally made it to the main game loop. Interestingly enough, I found all my games take about half the lines for the setup and the other half for the game loop. Defender is not so different - the game loop starts in the middle of 5th line. I have REM statements on many of the lines and will only add expanded comments where they add.

```
REPEAT
  TT=T0-TIME : rem countdown timer
  POKE 657,6 : rem position cursor for time display
  ?TT;A$;
  POKE 657,13 : rem position cursor for VICTIMS and SCORE display
  ?PEOPLE;A$;SCORE;

  REM PROCESS FIRE BUTTON
  F0=F
  F=STRIG(0)
  FIRE=F0&(1-F) : REM DISABLE RAPID FIRE
  FY=PY*FIRE+FY*(1-FIRE) : REM REMEMBER Y-LOCATION OF LASER
  FOR I=1 TO 3
    POKE PM+I*$100+$08+FY,FIRE*$FF : REM ADD OR REMOVE LASER IN
PLAYER : REM DP. TAKES 1 LESS CHAR THAN POKE : +$40 IS BAKED INTO
PM
    POKE $D000+I,112+10*DX+18*DX*I :  REM POSITION LASER
    POKE 704+I,8+16*RAND(16) : REM SET A RANDOM COLOR
  NEXT I
  REM PROCESS HITS and COLLISIONS - combine together to avoid
duplicate commands
```

This is the first bit of AI that I combined together to reduce the amount of code. I hope I can remember what is does!

First, check to see if either the firebutton was pressed of if there is a player-playfield collision.

```
  IF PEEK($D004)&4+FIRE
```

Once we're in, why we are in is stored in I

```
    I=FIRE
```

When the lasers are fired, we have to loop to clear out the bad guys.

```
    REPEAT
```

The subject location is computed and stored in FPOS. Note the variable I is now being used as a counter.

```
      FPOS=TRUNC(LINEONE+RDRY*FIELDWIDTH+XWLD+10.5+I*DX)
```

Here the item in screen memory is grabbed. Only the lower 4 bits matter as the upper four indicate color.

```
      WHAT=PEEK(FPOS)&$0F
```

Check to make sure we don't shoot the mountains.

```
      IF WHAT>3
```

These next two lines were repeated a lot in the original code. Having to update both the playfield and the radar display takes space!

```
        'erase characters
        POKE FPOS,0
        C.0 : PLOT QUARTER + ((FPOS-LINEONE) MOD
FIELDWIDTH)/2,RDRY+1
        'if hit a bad guy...
```

All the bad guys are characters > 4. I suppose a latent adbductor==9 could be destroyed, but I've tried and it's probably unlikely to occur.

```
        IF WHAT>4
          SCORE=SCORE+100
          IF I=0
            'collided with a bad guy!
            POKE $D000,0
            LIVES=LIVES-1
            POKE $BF62+LIVES,0 : REM DP. TAKES 1 LESS CHAR THAN POKE
            VX=0
            PAUSE 60
            POKE $D000,120
          ENDIF
        ELSE
          'OOPS! hit a citizen!
          PEOPLE=PEOPLE-1
```

If the citizen was being abducted, then turn the abductor $88 into a static bad guy $85.

```
          IF PEEK(FPOS-FIELDWIDTH)=$88
            POKE FPOS-FIELDWIDTH,$85
          ENDIF
        ENDIF
      ENDIF
    I=I+1
```

If we entered the loop because of a collision, the I was 0 and is now 1. That will exit. If we entered because we fired, then I was 1 and is incremented until I=10.

```
    UNTIL (I=1) ! (I=10) : REM OR IMOD9=1
  ENDIF
  POKE $D01E,1 : REM HITCLR

  REM PROCESS STICK INPUT
  S=STICK(0)
```

```
  UD=(S&2=0)*(PY<145)-(S&1=0)*(PY>0)
  PY=PY+4*UD : REM LIMITS 46144 , 46288 B440-B4D0
  LR=(S&8=0)-(S&4=0) : REM CREATE +/1 VALUES FOR LEFT/RIGHT
```

I use an IIR discrete time filter to create acceleration/deceleration effects.

```
  VX = VX/2 + LR/2 : REM USE IIR FILTER FOR ACCELERATION/DRAG
EFFECT
  DX = DX*(LR=0)+LR : REM DIRECTION INDICATOR
  XWLD = (XWLD + VX + 139) MOD 139 : REM UPDATE HORIZONTAL POSITION
IN WORLD COORDINATES
```

Because the X coordinate is floating point, I can do both coarse and fine scrolling. The fraction and truncation commands are taken advantage of here:

```
  XF=FRAC(XWLD)*8 : REM FINE SCROLL VALUE
  XT=LUT+TRUNC(XWLD)*60 : REM COARSE SCROLL VALUE
```

The missile portion of the PM graphics are used to indicate the location on the radar. These few lines move the ship.

```
  REM MISSILE SPRITES FOR RADAR POSITION INDICTATOR
  POKE $B32A+RDRY,$90 : REM RADAR INDICATOR OLD POSITION
  RDRX = TRUNC(XWLD/2)+87 : REM CONVERT WORLD COORDINATE TO RADAR
COORDINATE
  RDRY = TRUNC((PY+6)/8) : REM RADAR Y FOR SHIP -6 TO 146 -> 0 TO
19
  POKE $B32A+RDRY,$91 : REM PUT SHIP ON RADAR
```

Next we update the display list, but wait for a vertical blank to avoid flicker. I tried this on real hardware and it seems to work well.

```
  PAUSE 0 : REM WAIT FOR VERTICAL BLANK TO REDUCE FLICKER
  POKE $D404,12-XF : REM FINE SCROLL
  MOVE XT,$A070,60 : REM COPY IN DL FOR COARSE SCROLL
  DPOKE $D006,257*RDRX+11 : REM MOVE INDICATORS IN RADAR DISPLAY
  'POKE $D007,RDRX
  POKE $D004,RDRX+5 : REM MOVE THE SHIP IN THE RADAR DISPLAY
  MOVE $B205-5*DX,PM+PY,14 : REM PUT SHIP ON THE SCREEN (+$40 IS
BAKED INTO PM)
```

This last set of conditionals processes the AI behaviors. Combining multiple cases into one logic flow was critical to fitting in the 10 lines. Here goes...

```
  REM PROCESS AI
  REM PROCESS PEOPLE THEN ABDUCTORS
```

There are abductors and citizens. Using the FOR loop, I process the citizens first, then the abductors.

```
  FOR I=1 TO 2
```

The pointer to screen memory location is retrieved

```
  LL=AI(IDX,I)
```

If it is valid, then proceed. Otherwise, it must have been destroyed earlier.

```
    IF LL>0
      'something is there
```

Find out what the pointer is pointing to.

```
      PLL=PEEK(LL)
```

If it is a latent abductor $09, then flip a coin to see if it appears $87.

```
      IF PLL=9
        'if waiting abductor, maybe it now appears
        IF RND<.5
```
The screen code $87 is used to indicate a descending abductor.
```
          POKE LL,$87
        ENDIF
      ELSE
```
If it wasn't an abductor, maybe it's something else
```
        IF PLL<>5
          'if not a static bad guy ...
```
Look below the subject
```
          LLP=LL+FIELDWIDTH
          PLP = PEEK(LLP)
```
And above the subject
```
          LLM = LL-FIELDWIDTH
```
Erase the subject so we can redraw it after moving it. Combining all the AI logic here avoids having to repeat these commands several times.
```
          X = QUARTER + ((LL-LINEONE) MOD FIELDWIDTH)/2
          Y = 1 + (LL-LINEONE) DIV FIELDWIDTH
          COLOR 0 : PLOT X,Y
```
The screen code $88 indicates an ascending abductor that must have a kidnap victim.
```
          IF PLL=$88
            'if an ascending abductor it must also have a citizen
```
Erase the kidnap victim on the radar.
```
            PLOT X,Y+1 : REM IS COLOR 0
```
Find out if the abductor is still going up or maybe can escape at the top:
```
            IF LLM>LINEONE
              'still going up
```
Erase the citizen from the play field, move the ship up one line.
```
              POKE LLP,0
              POKE LLM,PLL
```
Remember where the abductor is.
```
              AI(IDX,2)=LLM
```
Replot the citizen below the ship.
```
              POKE LL,4
```
Remember where the citizen is.
```
              AI(IDX,1)=LL
```
Redraw them on the radar.
```
              COLOR 1 : PLOT X,Y
              COLOR 3 : PLOT X,Y-1
            ELSE
```
If the abductor escapes, erase them, set their locations to invalid, and reduce the citizen count.
```
              'escaped with a citizen!
              POKE LLP,0
              POKE LL,0
              AI(IDX,1)=-1
              AI(IDX,2)=-1
              PEOPLE=PEOPLE-1
```

```
          ENDIF
        ENDIF
```
If the subject is a citizen or descending abductor, deal with them now
```
        IF (PLL=4)!(PLL=$87)
          IF PLP=4
```
If there a citizen below the abductor, then kidnap them!
```
            'descending abductor
            POKE LL,PLL+1
          ELSE
            'falling citizen who was rescued or descending
abductor!
```
This logic was tricky and I only figured it out by working out the individual one separately at first. If there's space below the subject and either space above the subject or the subject is an abductor then...
```
            IF (PLP=0)&( (PEEK(LLM)=0)!(PLL=$87) )
```
Erase the subject and move it down one line. That works for either one.
```
              POKE LL,0
              POKE LLP,PLL
```
Update the location of the subject.
```
              AI(IDX,I)=LLP
              Y=Y+1
            ENDIF
```
Put the subject back on the radar. The subject is $87, then use color 3.
```
            COLOR 1 + (PLL&2) : PLOT X,Y
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  ENDIF : REM LL>0
NEXT I
```
Move onto the next set of AI's
```
IDX=(IDX+1) MOD NA
```
And keep on going until out of time or the hero is dead.
```
UNTIL (TT<0) ! (LIVES=0)
POKE 657,2
?"game over";
```
That's it!

Thanks for reading! I hope you get a chance to play and have fun.