

Chapter 10 - Tile and Sprite Layers from BASIC

Several picture chips build a screen from small repeatable elements instead of asking you to draw every pixel. This chapter is a BASIC cookbook for those tile and sprite patterns. The full register maps remain in each chip chapter; this chapter shows typed programs that can be used as starting points on the same shared bus machine.

10.1 The two ideas

Tiles are small bitmaps, usually 8x8, repeated across a grid. Each cell names a character or tile and a colour or attribute. Tiles save memory because the same glyph can appear many times.

Sprites are moving objects placed independently of the tile grid. Intuition Engine has one classic sprite system: GTIA players and missiles. VideoChip and Voodoo do not have dedicated sprites; use the VideoChip blitter or Voodoo textured quads for that job.

Chip	Tile-style path	Sprite-style path
VideoChip	Bitmap framebuffer only	Blitter copy, masked copy, alpha copy, scale
VGA	80 by 25 text cells in SCREEN 3	None
TED video	40 by 25 character cells and colour RAM	None
ANTIC/GTIA	Display-list text and bitmap rows	4 players and 4 missiles
ULA	32x24 attribute cells over bitmap	None
Voodoo	None	Two textured triangles per quad

10.2 VGA text cells

Purpose: VGA text mode is the quickest way to put a dense status or menu layer on screen. Setup is SCREEN 3. Each cell is two bytes at \$B8000: character code first, attribute second. The attribute low nibble is foreground, bits 4 to 6 are background, and bit 7 is blink.

This example writes directly into the text cell memory:

```
10 REM VGA TILE BANNER
20 SCREEN 3
30 COLOR 15,1
40 CLS
50 DATA 86,71,65,32,84,73,76,69
60 FOR I=0 TO 7
70 READ C
80 A=&H000B8000+(10*80+36+I)*2
90 POKE8 A,C
100 POKE8 A+1,&H1E
110 NEXT I
```

Expected result: the words VGA TILE appear near the middle of the screen in yellow on blue. Direct writes affect only the addressed text cells; CLS and PRINT still use the same text buffer.

The useful address is on line 80. A row has 80 cells and each cell is two bytes, so $(row*80+column)*2$ finds the character byte. Line 90 writes the ASCII code, and line 100 writes the colour attribute. Change &H1E to &H4F for white letters on a red background.

10.3 TED character cells

Purpose: TED text mode gives a 40 by 25 playfield with a character byte and a parallel colour byte per cell. Setup is TED ON, display enable in TED_V_CTRL1, and 40-column selection in TED_V_CTRL2.

Default text memory starts at \$F3000. Colour RAM starts at \$F3400. Both are byte arrays.

```
10 REM TED CHARACTER GRID
20 TED ON
30 POKE8 &H000F0F20,&H18
40 POKE8 &H000F0F24,&H08
50 POKE8 &H000F0F30,&H06
60 POKE8 &H000F0F40,&H5E
70 FOR I=0 TO 999
80 POKE8 &H000F3000+I,32
90 POKE8 &H000F3400+I,&H71
100 NEXT I
110 T$="TED GRID"
120 FOR I=1 TO LEN(T$)
130 A=&H000F3000+12*40+16+I
140 C=&H000F3400+12*40+16+I
150 POKE8 A,ASC(MID$(T$,I,1))
160 POKE8 C,&H10+I
170 NEXT I
```

Expected result: a TED text screen with a coloured message. The important side effect is that the character byte and colour byte are separate; changing one does not alter the other.

The first loop clears both TED planes: \$F3000 receives spaces and \$F3400 receives a starting colour byte. The second loop writes the message into the same cell positions in both planes. The +I in line 160 deliberately changes the colour byte for each character so the parallel colour RAM is easy to see.

10.4 ANTIC display-list tiles

Purpose: ANTIC display lists let one screen mix several tile and bitmap modes. Setup needs a display list, screen bytes for that list to fetch, GTIA colours, then ANTIC MODE with display-list DMA enabled.

This example builds a 40 column checkerboard in mode 2:

```

10 REM ANTIC CHECKERBOARD TILES
20 DL=&H0200:SCR=&H0300:CH=&H0800
30 FOR R=0 TO 7
40 POKE8 CH+8+R,&HFF
50 NEXT R
60 POKE8 DL+0,&H42
70 POKE8 DL+1,SCR AND 255
80 POKE8 DL+2,INT(SCR/256)
90 FOR I=0 TO 22
100 POKE8 DL+3+I,2
110 NEXT I
120 POKE8 DL+26,&H41
130 POKE8 DL+27,DL AND 255
140 POKE8 DL+28,INT(DL/256)
150 FOR Y=0 TO 23
160 FOR X=0 TO 39
170 POKE8 SCR+Y*40+X,(X+Y) AND 1
180 NEXT X
190 NEXT Y
200 ANTIC CHBASE INT(CH/256)
210 GTIA COLOR 0,&H04
220 GTIA COLOR 1,&H9A
230 GTIA COLOR 4,&H00
240 ANTIC DLIST DL
250 ANTIC MODE &H22
260 ANTIC ON

```

Expected result: a blue-on-black checkerboard. Display-list addresses are 16-bit. This listing keeps the display list, screen bytes, and character bytes below \$1000 so they are safely below BASIC's own program and variable work area.

The display list starts with a load-memory-scan instruction at line 60; lines 70 and 80 give ANTIC the low and high bytes of SCR. Lines 90-110 repeat mode 2 for the remaining rows, then lines 120-140 make the list loop back to itself. The nested loop writes the tile numbers that ANTIC fetches while GTIA supplies the colours.

10.5 ULA bitmap cells

Purpose: the ULA does not store character codes. It stores a bitmap plus a 32x24 grid of attribute bytes. Each attribute colours the 8x8 bitmap cell beneath it.

```

10 REM ULA ATTRIBUTE TILES
20 ULA ON
30 ULA CLS &H00
40 FOR Y=0 TO 23
50 FOR X=0 TO 31
60 A=((X+Y) AND 7)+64
70 ULA ATTR X,Y,A
80 NEXT X
90 NEXT Y
100 FOR I=0 TO 191
110 ULA PLOT I,I
120 ULA PLOT 255-I,I
130 NEXT I

```

Expected result: two diagonal bitmap lines crossing a grid of changing attribute colours. The side effect to remember is that changing an attribute recolours all set bitmap pixels inside that 8x8 cell.

This is the ULA idea in two passes. The first pass writes only the attribute map, one byte per 8x8 cell. The second pass writes bitmap pixels. Because the ULA combines those two memories at display time, the same diagonal bitmap bits take their colours from whichever attribute cell they pass through.

10.6 GTIA players and missiles

Purpose: GTIA players and missiles are the dedicated sprite path. Players are 8 pixels wide; missiles are 2 pixels wide. GTIA PLAYER and GTIA MISSILE set horizontal position. GTIA GRAFP and GTIA GRAFM supply direct graphics bytes. GTIA GRCTL 3 enables player and missile graphics.

This listing records a player and a missile down the visible frame:

```
10 REM GTIA PLAYER AND MISSILE
20 DL=&H0200:SCR=&H0300:CH=&H0800
30 FOR R=0 TO 7
40 POKE8 CH+8+R,&HFF
50 NEXT R
60 POKE8 DL+0,&H42
70 POKE8 DL+1,SCR AND 255
80 POKE8 DL+2,INT(SCR/256)
90 FOR I=0 TO 22
100 POKE8 DL+3+I,2
110 NEXT I
120 POKE8 DL+26,&H41
130 POKE8 DL+27,DL AND 255
140 POKE8 DL+28,INT(DL/256)
150 FOR Y=0 TO 23
160 FOR X=0 TO 39
170 POKE8 SCR+Y*40+X,(X+Y) AND 1
180 NEXT X
190 NEXT Y
200 ANTIC CHBASE INT(CH/256)
210 GTIA COLOR 0,&H04
220 GTIA COLOR 1,&H9A
230 GTIA COLOR 4,&H00
240 GTIA COLOR 5,&H46
250 GTIA COLOR 7,&HCE
260 GTIA GRCTL 3
270 GTIA PLAYER 0,110,1
280 GTIA MISSILE 2,190
290 FOR Y=0 TO 191
300 GTIA GRAFP 0,&H3C
310 GTIA GRAFM 4
320 POKE32 &H000F2120,0
330 NEXT Y
340 ANTIC DLIST DL
350 ANTIC MODE &H2E
360 ANTIC ON
```

Expected result: a checkerboard playfield with a red player column and a green missile column. Writing ANTIC_WSYNC at \$F2120 advances the scanline capture, so the same direct graphics byte is recorded on each line.

Most of the listing is the same ANTIC checkerboard setup used in the previous example. The new part begins at line 240: colours 5 and 7 are the player and missile colours, GRCTL 3 enables direct object graphics, and the FOR Y loop records one player byte and one missile byte on each captured scanline.

10.7 Voodoo textured quads

Purpose: a Voodoo sprite is a screen-aligned quad drawn as two triangles. The texture supplies the sprite picture; alpha, chroma key, and blending can be added with the pipeline commands from Chapter 9.

This example uploads a 2x2 texture and maps it onto a 64x64 quad:

```
10 REM VOODOO TEXTURED QUAD
20 VOODOO ON
30 VOODOO DIM 640,480
40 POKE32 &H000F8110,&H00008200
50 VOODOO CLEAR &HFF101010
60 POKE32 &H000D0000,&HFF0000FF
70 POKE32 &H000D0004,&HFF00FF00
80 POKE32 &H000D0008,&HFFFF0000
90 POKE32 &H000D000C,&HFFFFFFF
100 TEXTURE DIM 2,2
110 TEXTURE MODE &H0A61
120 TEXTURE UPLOAD
130 TEXTURE ON
140 VOODOO COMBINE 1
150 VERTEX A 160,120
160 VERTEX B 224,120
170 VERTEX C 160,184
180 POKE32 &H000F8088,0
190 VOODOO TRICOLOR 4096,4096,4096,4096
200 VOODOO TRIUV 0,0,0,0,0,0
210 POKE32 &H000F8088,1
220 VOODOO TRICOLOR 4096,4096,4096,4096
230 VOODOO TRIUV 262144,0,0,0,0,0
240 POKE32 &H000F8088,2
250 VOODOO TRICOLOR 4096,4096,4096,4096
260 VOODOO TRIUV 0,262144,0,0,0,0
270 TRIANGLE
280 VERTEX A 224,120
290 VERTEX B 224,184
300 VERTEX C 160,184
310 POKE32 &H000F8088,0
320 VOODOO TRICOLOR 4096,4096,4096,4096
330 VOODOO TRIUV 262144,0,0,0,0,0
340 POKE32 &H000F8088,1
350 VOODOO TRICOLOR 4096,4096,4096,4096
360 VOODOO TRIUV 262144,262144,0,0,0,0
370 POKE32 &H000F8088,2
380 VOODOO TRICOLOR 4096,4096,4096,4096
390 VOODOO TRIUV 0,262144,0,0,0,0
400 TRIANGLE
410 VOODOO SWAP
```

Expected result: a four-colour square appears at (160,120). The texture words use little-endian RGBA packing, as described in Chapter 9.

The texture is only four pixels, loaded by lines 60-90. Lines 150-270 draw the first triangle of the square, and lines 280-400 draw the second. Each vertex gets both a screen position and a texture coordinate. `V00D00 SWAP` publishes the completed frame. Change the four texture words to recolour the sprite without changing the triangle geometry.

10.8 Combining layers

The compositor stacks the picture chips automatically. This gives a useful pattern: keep the background on one chip and put the moving or status layer on another.

Combination	Useful for	Limits
Voodoo with VGA text below	Scoreboards, menus, debug readouts in transparent areas	VGA is layer 10; Voodoo is layer 20, so Voodoo covers VGA where it draws.
TED tiles with ANTIC players	Tile playfield with hardware moving objects	TED is layer 12; ANTIC/GTIA is layer 13, so players appear above TED.
ULA bitmap with ULA attributes	Cheap colour animation	Attribute changes affect whole 8x8 cells.
VGA text above VideoChip blitter sprites	Software sprite copies with a text/status layer	VideoChip is layer 0; VGA is layer 10, so VGA covers VideoChip where it draws.

Turning on more than one chip is allowed. The chips share the bus, but not each other's register sets or private video memory. They also share the final compositor, so layer order decides which visible pixel wins.

A useful whole-machine exercise is to combine one example from this chapter with one from a chip chapter. For example, use the TED character grid from section 10.3 as a low layer, then add the GTIA player column from section 10.6 above it. The two chips do not share video RAM, but they do share the final compositor, so the object layer appears over the tile layer without a copy step.

10.9 Checklist

For each tile or sprite layer:

1. Enable the chip.
2. Choose the picture mode.
3. Fill the tile, bitmap, texture, or player data.
4. Write colour or attribute data.
5. Set positions, display-list pointers, or quad vertices.
6. Use the chip's publish step if it has one, such as `V00D00 SWAP`.

The audio chapters begin next.