BRITAIN'S LEADING MAGAZINE FOR AMIGA OWNERS

AMIGA FORMAT

ISSUE 8 / MARCH 1990 / £2.95

Fast Forward Into Video

Take part in the next Amiga revolution

COVERDISK 8

X-OUT

THE PLAYABLE DEMO OF RAINBOW ART'S MANIC SHOOT-EM-UP

NO AMIGA COVERDISK?
DEMAND ONE FROM YOUR NEWSAGENT NOW!

UNMISSABLE GAMES REVIEWS

PLUS REVIEWS OF PAGESETTER 2, CROSSDOS, ULTRACARD, MASTER SOUND, GFA BASIC COMPILER, HISOFT EXTEND

03
9 770957 486004

DAVE JONES, programmer of Psygnosis' hits *Menace* and *Blood Money*, presents part two of his series in which he divulges the tricks

THE WHOLE TRUTH ABOUT GAMES PROGRAMMING: 2

of the trade used by top games programmers.

This month:

SCROLLING

This month's example with source is the dual playfield *Menace* scroll. The framework source form last month has been used to allow the scroll to be executed, with return to the CLI upon pressing the left mouse button. Try executing the assembled file on the disk, what you see should hopefully be recognisable as *Menace*, minus any aliens or your ship on the screen.

When designing your scroll routine there is one major decision to make: namely, should it be a hardware or software scroll? First I'll explain the differences.

Hardware Scroll

The Amiga has the ability to hardware scroll the display screen. This means the entire display can be shifted pixelly, left or right, with virtually no overhead or processor usage. It actually does better than this in that it can change the scroll value every line if required: take a look at *Shadow Of The Beast* for some impressive use of the hardware scroll.

Software Scroll

A software scroll entails using the processor, or preferably the blitter, to physically shift the display memory the required number of pixels. Take for example a typical 32-colour screen that requires 40000 bytes: to scroll the entire display memory, even using the blitter, would take the best part of a frame (1/50th of a second).

Pros and Cons

It seems fairly obvious at first glance that the hardware scroll is the one to go for: however, thoughts must now turn to what exactly will be drawn into the display memory. To move an alien about the screen for either method requires a simple procedure as follows...

1. Save the memory where the alien is to be drawn.
2. Draw the alien (masked) into this memory.
3. When moving the alien, restore the memory and go back to (1).

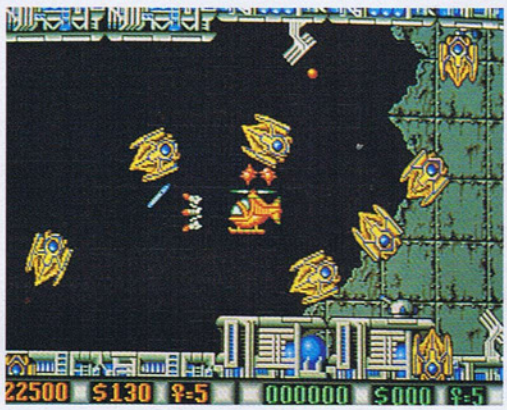
If we did not do the saving and restoring of the display memory, then as we moved that alien, a 'trail' of itself would be left when it moved. The above procedure is exactly what happens in *Menace*, where the saving and restoring does take up a major part of the execution time of the game. This is where using the software scroll can have an advantage. With the software scroll the usual method is to use the blitter to copy the display memory, shifting it as it goes, to another part of memory, which will obliterate the contents of what was previously there. This means that only step 2 of above need be executed when moving aliens about, as the whole of the display memory is restored while it is being scrolled.

If you envisage having a LOT of objects flying about on a scrolling screen, then there comes a point where the software scroll will save you more execution time than the hardware one. The software one is also simpler to write, not having to bother with steps 1 and 3 above. Incidentally in *Blood Money* I switched to a software scroll for the very reason of the number and size of the aliens kicking about compared to those flying around in *Menace* There are, of course, many variations on scrolling techniques which are dreamed up by programmers: it is simply a case of sitting down with pen and paper and working out which one is best suited to your own game.

Scenery Blocks

With the scrolling method decided upon I had to come up with a technique for scrolling through approximately 30 screens for one *Menace* level. The simplest way would be to have 30 screens laid end to end in memory and simply hardware scroll through memory. However, at approximately 24 Kbytes per screen this would require some 720 Kbytes, not exactly easy with only 512 Kbytes! Game playing areas therefore tend to be made up from maps.

The scenery graphics were broken up into 16x16 blocks, each of these given a number from 0-255 (to store as a byte). To make best use of the blocks many blocks were designed to fit together in certain ways giving as much variety as possible. Some games that use this technique are easily spotted when graphic ▸



In one of David's other games, *Blood Money*, he used a software scroll to enable him to put more objects on the screen – as you can see from this screenshot of some furious action.

blocks that do not quite match up are placed together – *Battle Squadron* exhibits this quirk. As *Menace* is a dual playfield game the maximum number of colours per block is 8, made up from 3 planes. Each block required 96 bytes of memory (2 bytes wide x 16 high x 3 planes) with a complete level taking 24576 bytes (256 blocks x 96 each).

The scrolling technique devised allowed us to scroll through an infinite number of screens, but required memory for only twice that of a normal screen. The *Menace* screen was larger than the normal 320 wide to make the playing area that bit larger. 16 pixels were added either side, expanding it to 352 pixels in length and providing a nice overscan effect. Another extra 16 pixels were also required at the left side due to the way the Amiga accomplishes the hardware scroll (these are the extra pixels that are normally hidden but are hardware scrolled on) – this is fully explained in the Amiga hardware manual. The actual size is therefore 368 pixels wide of which 352 are displayed. As mentioned, the scroll routine requires memory for two screens laid side by side (see figure 1), we can calculate the memory required as...

46 bytes wide
x 2 screens
x 192 high
x 3 planes
= 52992 bytes

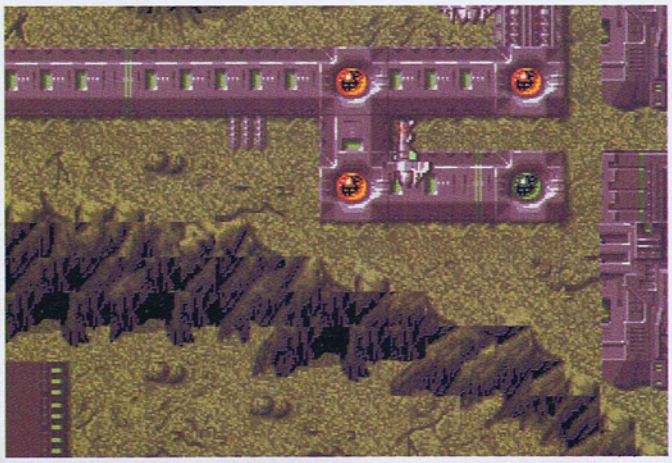
The 192 line height of the playing area was chosen as it is the closest multiple of 16 to 200, the game panel adds another 32 pixels to the overall height bringing the full screen size to 224 pixels. The background playfield is constructed in a similar way (see figure 2) but requires an extra 32 pixels at the end of each screen for clipping purposes (more about this at a later stage). The memory required for the background is...

50 bytes wide
x 2 screens
x 192 high
x 3 planes
= 57600 bytes

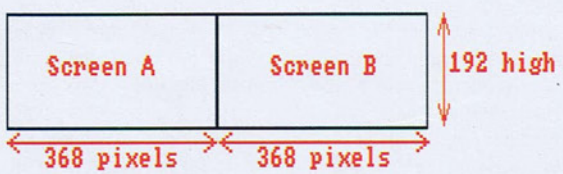
Given that that one screen is 368 x 192 pixels, this corresponds to 23 x 12 blocks (each block being 16x16). As each block is stored as a byte in the map, then map data for one screen would be 23 x 12 = 276 bytes. For approximately 30 screens per level the map data would therefore be some 8280 bytes. Looking at the size of the file MAP on the disk, which is the map data for level 1 of *Menace*, shows a file size of 5282 bytes – so level 1 consists of roughly 19 screens. The map data in this file is simply organised as 'strips of bytes'. This means that every 12 bytes (the number of blocks high the screen is) represent the 12 graphic blocks that sit one on top of another to form a 16 x 192 high strip which is scrolled on from the right.

That actual graphic data for each block is stored in the file FOREGROUNDS. As discussed, each block is 96 bytes in length, given that the foregrounds file on the disk is 24480 bytes in length, we know this will contain 255 graphic blocks (1 less than the 256 maximum allowed). The first 96 bytes are always 0, as block 0 is a special case being a blank block (there has to be some blank areas on the screen to fly through!).

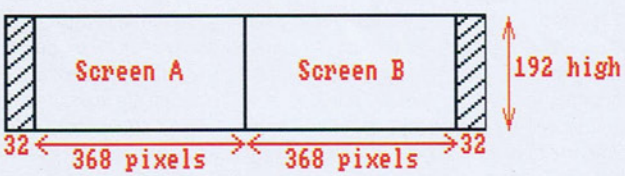
You could try experimenting with your own graphic data and map. If you altered the bytes in the map in any way, then you will see 16 x 16 blocks scrolling on that were obviously not designed to fit together. You can even try changing the map file to some other file, as it is simply a sequence of bytes that can be any value. The program will not crash doing this. You can even do this with the foregrounds file to produce some pretty random graphics!.



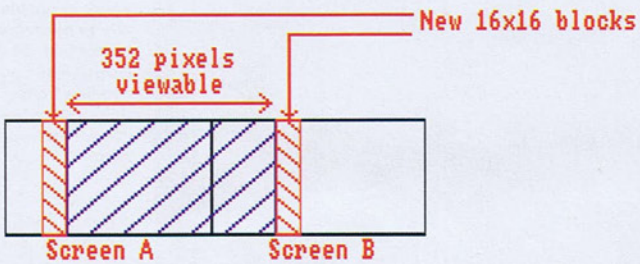
***Battle Squadron* from Electronic Zoo exhibits a programming quirk where graphic blocks that do not quite match up have been used. Can't see it? Then look closely at the ridge running across from the left of the screen.**



Scenery (foreground) screen
Figure 1



Background screen
Figure2



Scenery (foreground) screen
Figure 3

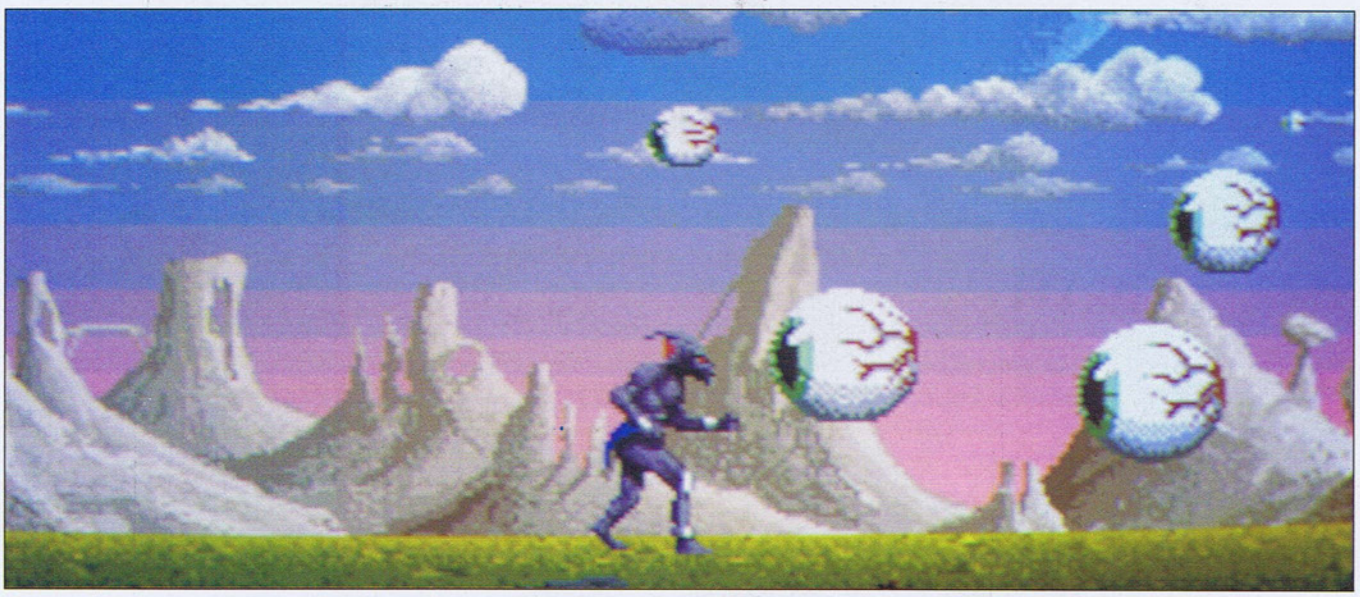
How the scroll works...

Now we know how the map and graphics are organised I will attempt to explain how the scroll works. If it sounds confusing, which it probably will at first, persevere, as when it clicks it should seem pretty straightforward.

Take a look at figure 3. This shows our two screens laid side by side in memory. At any one time we are displaying 352 pixels (22 words) of this data. The bit plane pointers on the Amiga can be positioned on any word (16 pixel) boundary. Incrementing the pointers therefore would scroll through memory 16 pixels at a time, which is a mega speed compared to the single pixel *Menace* requires. We therefore use the hardware scroll to shift the display pixelly from 0 to 15, then when we want to scroll to the 16th position we increment the bit plane pointers but reset the hardware scroll back to 0. We will carry on doing this until we have scrolled entirely through screen A and are displaying screen B. At this point we reposition the bit plane pointers back to display screen A and repeat the procedure again. OK, this will smooth scroll us from A to B.

Now, to keep new data coming onto the screen we draw graphic blocks as defined in the map, one strip at a time (16 x 192 pixels) just to the right of where we are displaying (as shown in figure 3). Therefore for every 16 pixels we scroll on we draw a new strip from the map, scroll another 16 pixels, draw a new strip etc, etc. Remember that the strip is being drawn just to the right of where the display is, so we cannot see it being drawn, but only see it scrolling smoothly on.

Right, if you understand so far you may notice a quirk in that when we have fully reached screen B, we reposition the plane pointers back to screen A and start again. This sudden jump to screen A though will cause a complete new screen to appear showing what was previously in screen A. This is where we apply the twist in the tail. As we are drawing the strips into screen B and scrolling them on, at exactly the same time we draw the same strip into screen A, just to the left of where we are displaying (see figure 3 again). This means that as we are forming and scrolling through screen B, the exact same data is being formed in Screen A, so when we are completely displaying screen B, screen A is also



***Shadow of the Beast*, another game from Psygnosis, demonstrates impressive use of hardware scrolling – which is the technique used in Menace.**

exactly the same. NOW when we display screen A again, nothing will seem to happen as the same data is being displayed, but we have moved the plane pointers back to screen A, allowing us to repeat this process, and to scroll through large numbers of screens with only two screens in memory!

If your brain has now turned to jelly with that lot, do not worry, the light will dawn soon. Read it a couple of times, remembering the problem you are trying to overcome.

The background playfield in *Menace* is scrolled through in the same way, although no map building is done as the background is a simple wrap scroll where whatever gets scrolled off on the left reappears again on the right. At the start of a level, the background screens A and B are both built identically from a small map that allows only 16 blocks maximum. The Background is scrolled once every SECOND frame to allow it to scroll half the speed of the foreground. This gives the nice parallax effect. The graphic data for these blocks are included as source in the scroll source on the disk. The background graphic blocks are only 4 colours.

The copperlist

Finally for this month a run down of the copperlist for the main game (Listing 1). I tend to put everything that describes the display into the copperlist, although many can simply be written with the 68000. It allows the full display to be quickly changed or referred to rather than looking through your source to find where you changed modulo's etc, for certain copperlists.

The first instruction is a 'wait for line 10', which simply allows a bit of time after a vertical blank occurs in which to change some values in the list, before the copperlist is executed again.

Next we set the bitplane pointers. Six planes in all for dual playfield, three for the back playfield (as defined first) then come the three for the front playfield. Note these all point to 0 as they will be initialised once we have allocated some screen memory.

Next come the control registers. BPLCON0 is set to six planes with dual playfield activated.

BPLCON1 sets both playfield scroll values to 15. As we want to scroll left we have to actually decrement the hardware scroll value, incrementing it will scroll us right.

BPLMOD's are set to the difference in width of the screens laid side by side in memory, to the displayed areas.

DDFSTRT and DDFSTOP are increased from the normal values by one word each, DDFSTRT is increased by a further word due to the hardware scroll. The hardware manual goes into this in greater depth.

DIWSTRT and DIWSTOP are set to reflect a screen size of 352 x 224 pixels. Note that the display is set higher up the screen than normal to allow 224 pixels to be viewed on an American system on which *Menace* appears as full screen with overscan.

COLOR and SPRITE registers are all set to 0 initially, these are set up by the initialisation routine of the game.

After 192 lines have been displayed a copper change occurs which switches the display to a 16 colour one in which the panel is displayed. The panel is 352 x 32 pixels, the graphic data is stored in the file PANEL on the disk. ■

LISTING 1 – MENACE COPPERLIST

```
clist           DC.W   $0A01,$FF00
copperlist      DC.W   bplpt+0,$0000,bplpt+2,$0000
                DC.W   bplpt+8,$0000,bplpt+10,$0000
                DC.W   bplpt+16,$0000,bplpt+18,$0000
                DC.W   bplpt+4,$0000,bplpt+6,$0000
                DC.W   bplpt+12,$0000,bplpt+14,$0000
                DC.W   bplpt+20,$0000,bplpt+22,$0000
                DC.W   bplcon0,$6600
scroll.value    DC.W   bplcon1,$00FF,bpl1mod,$0036
                DC.W   bpl2mod,$002E,bplcon2,$0044
                DC.W   ddfstrt,$0028,ddfstop,$00D8
                DC.W   diwstrt,$1F78,diwstop,$FFC6
colours         DC.W   color+0,$0000,color+2,$0000
                DC.W   color+4,$0000,color+6,$0000
                DC.W   color+8,$0000,color+10,$0000
                DC.W   color+12,$0000,color+14,$0000
                DC.W   color+16,$0000,color+18,$0000
                DC.W   color+20,$0000,color+22,$0000
                DC.W   color+24,$0000,color+26,$0000
                DC.W   color+28,$0000,color+30,$0000
                DC.W   color+32,$0000,color+34,$0000
                DC.W   color+36,$0000,color+38,$0000
                DC.W   color+40,$0000,color+42,$0000
                DC.W   color+44,$0000,color+46,$0000
                DC.W   color+48,$0000,color+50,$0000
                DC.W   color+52,$0000,color+54,$0000
                DC.W   color+56,$0000,color+58,$0000
                DC.W   color+60,$0000,color+62,$0000

sprite          DC.W   sprpt+0,$0000,sprpt+2,$0000
                DC.W   sprpt+4,$0000,sprpt+6,$0000
                DC.W   sprpt+8,$0000,sprpt+10,$0000
                DC.W   sprpt+12,$0000,sprpt+14,$0000
                DC.W   sprpt+16,$0000,sprpt+18,$0000
                DC.W   sprpt+20,$0000,sprpt+22,$0000
                DC.W   sprpt+24,$0000,sprpt+26,$0000
                DC.W   sprpt+28,$0000,sprpt+30,$0000

                DC.W   $DF01,$FF00
                DC.W   bplcon1,$0000,bplcon0,$4200,ddfstrt,$0030
rastersplit2    DC.W   bplpt+0,$0000,bplpt+2,$0000
                DC.W   bplpt+4,$0000,bplpt+6,$0000
                DC.W   bplpt+8,$0000,bplpt+10,$0000
                DC.W   bplpt+12,$0000,bplpt+14,$0000
colours2        DC.W   color+20,$0000,color+30,$0000
                DC.W   color+2,$0000,color+4,$0000
                DC.W   color+6,$0000,color+8,$0000
                DC.W   color+10,$0000,color+12,$0000
                DC.W   color+14,$0000,color+16,$0000
                DC.W   color+18,$0000,color+22,$0000
                DC.W   color+24,$0000,color+26,$0000
                DC.W   color+28,$0000,color+0,$0000
                DC.W   bpl1mod,$0000,bpl2mod,$0000
                DC.W   $DF01,$FF00,intreq,$8010
                DC.W   $FFFF,$FFFE
```