BRITAIN'S BEST SELLING MAGAZINE FOR AMIGA OWNERS

AMIGA FORMAT

ISSUE 9 / APRIL 1990 / £2.95

CAPTURE YOUR IMAGINATION

Make your dreams become reality in the world of Multimedia

COVERDISK 9 PIPE MANIA

[illegible]ABLE DEMO FROM EMPIRE OF THE MOST ADD[illegible]TIVE PLUMBING GAME EVER

NO AMIGA COVERDISK? DEMAND ONE FROM YOUR NEWSAGENT NOW!

More STUNNING games than EVER before

3 OUTSTANDING Format Golds

DIGI-VIEW 4 ■ MUSIC X JNR ■ X-SPECS 3D ■ DELUXE VIDEO III ■ PC EMULATOR ■ AMOS

DESIGNING THE MAIN SHIP

THE WHOLE TRUTH ABOUT GAMES PROGRAMMING: 3

Top programmer **DAVE JONES**, author of Psygnosis' hits *Menace* and *Blood Money*, reveals more secrets of the art. This month:

One of the most important things about an arcade-style game is the look and feel of the object the player is controlling. Ninety-nine percent of the player's attention will be focussed on controlling and watching this object, so any problems in the control method or any dire-looking graphics will soon put people off playing the game: so it's wise to put an awful lot of effort into movement and definition of the main character.

With *Menace*, we tried a few different spaceships before we found one that most people liked. The control method for moving a ship about the screen was, of course, to be a nice, simple eight-direction affair, because you can't really ask for too much variation in a scrolling shooter.

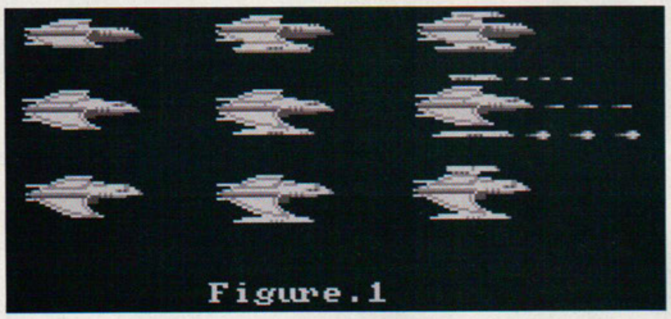
However, because we wanted to control the ship with the mouse as well as with the joystick, some inertia was added to the ship. This makes the mouse-controlled ship move more like a cursor would under mouse control.

The inertia is simply a snippet of code that prevents you instantly switching direction, and instead forces the ship to slow down in the direction it was going, stop, and then accelerate to its maximum speed in the chosen direction. It is not so noticeable on the initial speed of your ship, two pixels per frame, but try changing the speed in the source to, say, six pixels and then give the ship a test run. ➤

Shaping the Ship

This month's source adds the main ship and weapon code to last month's scrolling background. It was decided right at the start of writing *Menace* that the main ship should make use of the Amiga's hardware sprites. There are normally eight sprites available, each of which can be 16 pixels wide by any height in three colours. However, the wider-than-normal screen on *Menace* steals some DMA cycles from the sprite hardware allowing only six sprites to be displayed. This would seem to be enough for the main ship, if we allocated two sprites for the outriders, leaving four sprites for the main ship. So take a quick look at Figure 1. This shows the first ship we used in *Menace,* which, you have to admit, does look pretty dire! The restriction of three colours was detracting far too much from the main ship, making it look pale compared to the rest of the graphics.

The next step, then, was to use the sprite overlay technique that the Amiga allows, which basically means that two sprites can be combined as one but with 16 colours. This chopped us down to only three sprites maximum. By combining the outriders with the back of the ship as one sprite, and the front of the ship as another sprite, this left us with one free for use if we needed it (which in the end we did not). The result was the ship in Figure 3, which is the ship that appears in Menace. Figure 2 was another ship we tried, which was my favourite but the big publishers in the sky plumped for the other one, so I gave in...



PAINTING PROBLEMS

Many questions programmers receive are of the form 'How do you get graphics from *DPaint* (which most people use) into the game?' A lot of a project's time is devoted to writing programs that grab the graphics and store them in the desired format. For *Menace,* programs were written that converted brushes to hardware sprite format, blitter format and raw screen format. These programs all involved a common IFF reader, along with code to save out the graphics to a DOS file.

These type of utilities are essential in writing a game, and luckily they are appearing in many different guises in the public domain, which should help you to start. Eventually you should sit down and write a flexible conversion program that can generate ST or PC format graphics for any other versions of a game that may be required. For example, here is the menu for our own conversion program (written in C) that has been developed over the years:

```
PC-ilbm2raw v1.2 (c)1989 DMA Design

Usage: ilbm2raw [options] filename [output filename]

Options available are :-

    -A    Sets machine type to AMIGA
    -b    Sets Bit Plane(s) Ignore for any of the bit planes
          Followed by numbers between 0 & 7 to select which bit
          plane(s)
    -B    Switches OFF Body generation
    -c    Switches OFF Color generation
    -C    Switches ON CBM64 bit-mapped image generation
    -d    Switches ON diagnostics
    -E    Sets machine type to EGA 16 colour
    -g    Sets grid pick up operations, 16x16 graphics picked up
    -G    Sets machine type to CGA
    -i    Switches mask Inversion ON, masks become the NOT OR of
          all the planes
    -m    Switches on mask as an extra (last) plane
    -M    Followed by width (in pixels) for grid operation
    -N    Followed by height (in pixels) for grid operation
    -r    Switches On ROW major order for grid,
          (default is column major order)
    -S    Sets machine type to ST
    -s    Switches ON source generation
    -V    Sets machine type to VGA 256 colour
    -Z    Switches OFF the Zero Check for grid operations
          ie. All zero blocks are saved out in grid operations
```

As you can see, the list is quite comprehensive: this took a while to write, but now means we very rarely have to write graphic utilities, because working from IFF screens means we can convert to most machine graphic types.

Adding up the Anims

The boxes around the ship outline the actual size of the sprite that had to be stored in the game. At the back of the ships you will notice the spaces at the top and bottom of the box. This is where the outriders appear, attached to the ship. The ship can be viewed straight on, or tilting up or down. Each weapon added therefore required another three animations to be drawn.

There are also two extra weapons in the form of cannons and lasers, making a total of nine animations, plus the ship with both weapons attached bring us to the total shown of twelve animations. The outriders have a possible five directions but rather than store the animation for every possible combination (12x6 = 72 animations) of ship with outrider, the outriders are stored seperately and drawn into the extra space left at the back of the ship every frame in the game. This is provoked by the usual speed to memory trade-off.

Creating the Code

Now on to this month's source code from the Coverdisk. The source has the following functions implemented since last month:

- Inertia ship movement
- Overlayed hardware sprites
- Joystick read
- Mouse read
- x,y to hardware sprite coordinate conversion
- ship animation

The ship is 32 pixels wide, and will therefore need two hardware sprites as a hardware sprite is a maximum of 16 pixels wide. The ship, however, contains 16 colours which is only possible by overlaying hardware sprites, which brings the number used to four. Figure 3 shows the *Menace* ship with the size box drawn around it. The back of the ship is 44 pixels high to accomodate the outriders: the front of the ship is 22 pixels high. The file **ships.s** on the disk contains the hardware sprites in source format. In this file you will see labels named **ship1** up to **ship4.** These correspond to the following basic designs:

ship1 – basic ship, no weapons
ship2 – ship with cannons
ship3 – ship with lasers
ship4 – ship with cannons & lasers

Each ship also has three sets of data: **shipN.1, shipN.2** and **shipN.3,** where the **.1** is the ship tilting up; **.2** is the ship side on and **.3** is the ship tilting down. In the source you will see a DC.L 0 statement at the begining and the end of each piece of data for a hardware sprite. The one at the beginning will contain the two control words defined in the hardware manual that describe the sprite's x,y position along with overlay information. The long word 0 at the end signifies the end of the sprite. The way the control words are layed out is quite messy, with bits and bytes in awkard places. The routine in the source called 'xy to sprite' takes a normal x,y pixel position in a couple of registers and returns the long control word in the correct format. A small routine like this will always come in handy from project to project.

We can work out how many bytes a ship animation takes with the following method:

back of ship = 2 bytes wide * 44 high * 2 planes = 176 bytes + 2 long words (control) = 184 bytes

front of ship = 2 bytes wide * 22 high * 2 planes = 88 bytes + 2 long words (control) = 96 bytes

ship animation = (184+96)*2 (due to overlaying) = 560 bytes ➤

This figure of 560 bytes will crop up quite often in the source to calculate where a certain ship animation is. The ship animation routine for tilting the ship up and down works by storing the animation address for a particular ship's side-on view: when the joystick is pushed up or down another variable is set to either –560 or +560 (normally 0 for the side-on view) which automatically adjusts the animation that is viewed. Changing the animation address to the ship with cannons for example, will still tilt the canons up & down as the offset from the side-on view to tilting up or down is still +/- 560.

Reading the Input

The joystick/mouse is read every frame, and the ship moved at this rate. Using hardware sprites makes this very simple no matter what speed the game runs at. *Blood Money* runs every three frames, but the players' ships are updated every frame. This has the advantage that even if a game slows down occasionally the player can still zip about at the same speed, so the slow-down is much less noticeable. This is accomplished by making the ship movement integrated into a vertical blank interrupt routine. *Menace* does not require this as the game runs in a frame anyway.

The joystick read routine is quite simple, the basics being explained in the hardware manual. The mouse routine was included to emulate the joystick if a joystick was not available. It is not a true mouse read routine as it only checks if the mouse is being moved up/down/left/right. If so, it modifies the results the joystick routine returns, making it look as though the joystick had been pressed in a certain direction. This method does away with having a mouse/joystick option in the game as you can use either at any time. A full-blown mouse routine would return information on the direction and speed of the mouse, and it is not too difficult to modify the routine to do this if you require this in your own game.

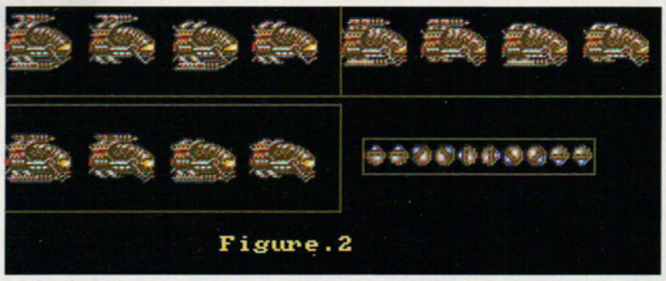


Figure.2

Making Motion

The 'moveship' routine is the main part of the ship code. Its main dealings are with the inertia on the ship. If, for example, you are moving right at three pixels at a time, you cannot simply press left and go left at three pixels at a time. A vector is used to gradually reduce and then increase your speed in the form +3,+2,+1,0,-1,-2,-3. This leads to a much more realistic feel to the movement on the ship. Small touches like this often make the game that bit more playable.

Tricks and Treats

Although only eight hardware sprites are usually available on the Amiga there are some tricks worth mentioning that can stretch this amount a little bit.

After a hardware sprite has been displayed it can be used to display some new data one scan line after the end of the last. For example, if the ship in *Menace* was at the top of the screen, then 45 pixels down (height + one scan line) we could draw the ship again if required on a different x position (or any y position > 45). This we could repeat all the way down until we ran out of space. The obvious drawback with this is that objects would always be in rows across the screen: they could not pass over each other vertically.

Other hardware sprites can cross over each other, though, so if you had some clever code that manipulated all eight sprites and sorted out sprites by saying 'This object here is further down the screen than this one, so I can re-use the same hardware sprite to display it, but this object has the same Y so it will require a different hardware sprite' you can in effect 'multiplex' sprites. In some instances you can multiplex 64 sprites down to the Amiga's eight depending on the restrictions you apply to their movement. This technique was extremely well used on the C64 and is now being used to some good effect on the Amiga. *Battle Squadron*, for example, uses hardware sprites for all the enemies' bullets and the players' firepower, which looks in excess of 32 sprites being displayed at once. ■

■ That's my ramblings over for this month. Back next month with some more juicy source.

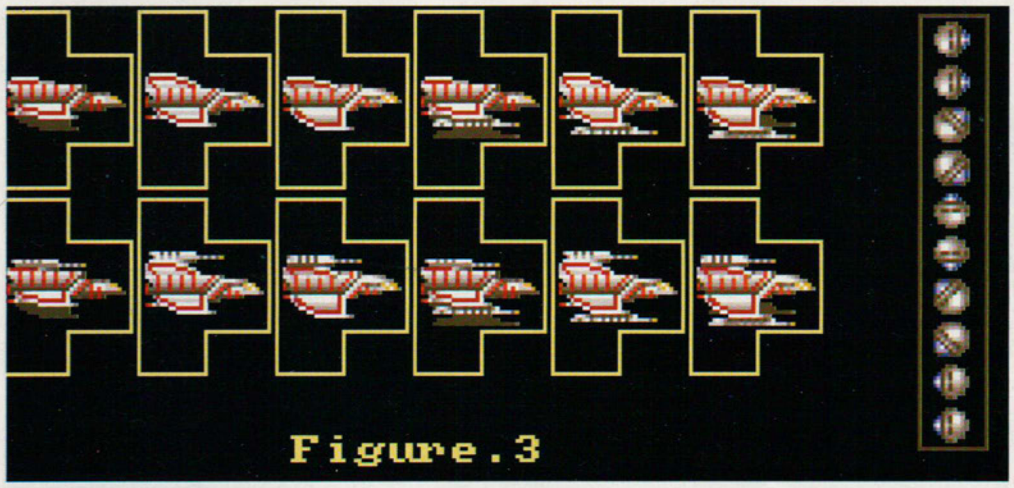


Figure.3