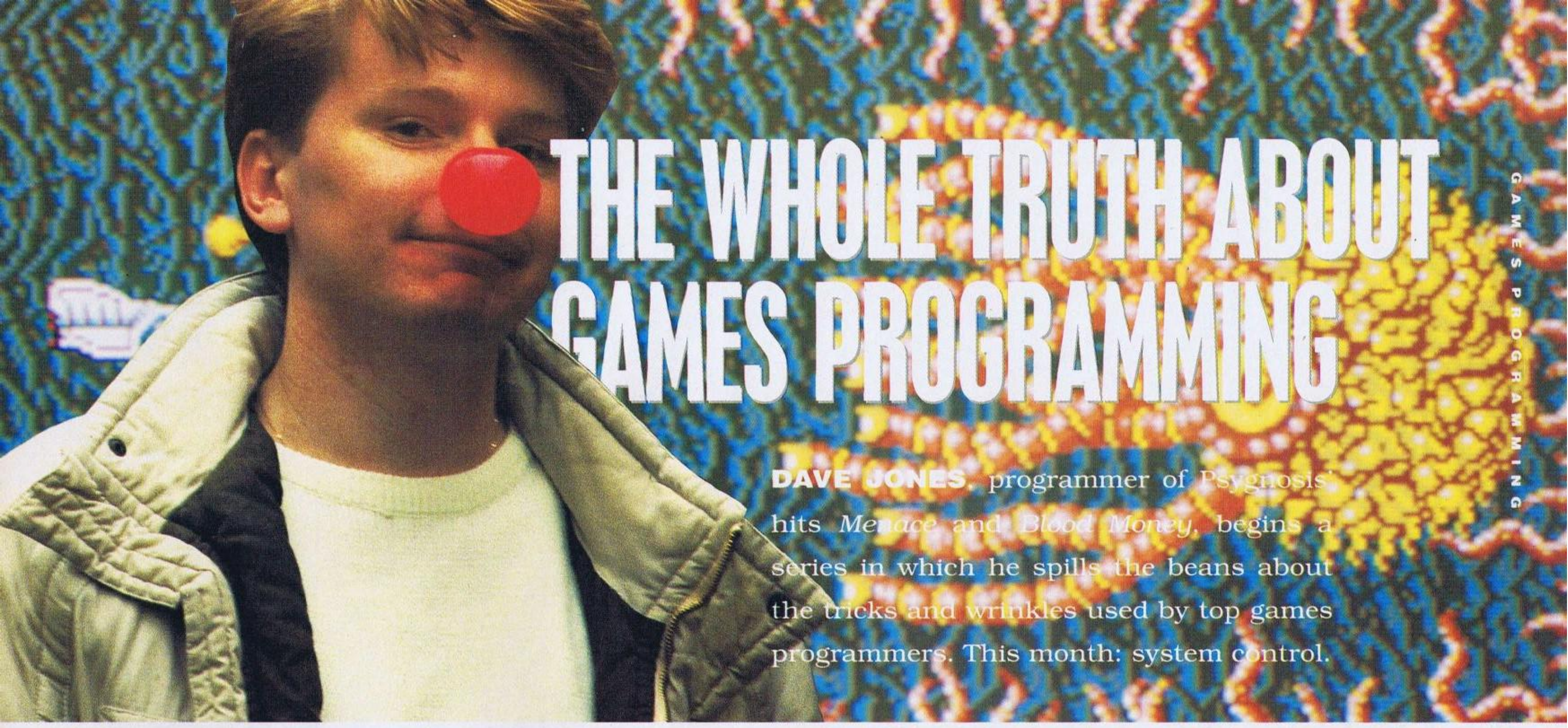


THE WHOLE TRUTH ABOUT GAMES PROGRAMMING

DAVE JONES, programmer of Psygnosis' hits *Menace* and *Blood Money*, begins a series in which he spills the beans about the tricks and wrinkles used by top games programmers. This month: system control.

In this series, Dave Jones will not only provide the real facts about how to program a best-selling game: he also intends to back it up by supplying the source code to his first great game, *Menace*. Each month, the Coverdisk will contain a piece of source code to illustrate the particular aspect of programming which Dave is discussing that month.

Usually, source code is one of the programmer's most jealously-guarded secrets, because it contains details of the tricks the author has learnt to make his code faster and more effective than that of his rivals. Very often, sections of the code are re-employed in later programs.

Of course, *Menace* is no longer a brand-new game and a remarkable amount has been learnt about programming the Amiga since Dave wrote it: so hopefully no harm will be done to Dave's personal prospects. But much of the information in these pages will be invaluable to anyone just starting out in programming who wishes to produce a seriously viable, up-to-date and saleable Amiga game.

Remember, this is serious stuff. The code contained on the Coverdisk is 68000 machine code, so some knowledge of the relevant language will be necessary before you can get on with writing your world-beating game. To use the code, you will need to assemble it using either *Devpac* from HiSoft, with which it was written, or Argonaut's *Argasm* as demoed on this month's Coverdisk. If you are using *Argasm*, be sure to include the extra piece of conditional code written by Jason. Good luck!

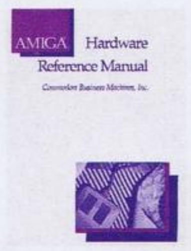
About Dave Jones...

Dave Jones is now 23 years old and lives in Dundee, Scotland. His first game, *Menace*, was released by Psygnosis in November 1988 to considerable acclaim from reviewers. It may look somewhat dated now, but many of the programming techniques it uses are extremely advanced.

Dave started work for Timex in Scotland when he left work, doing development work for the early Spectrums, a background which gave him a good insight into computer hardware. Although originally involved in writing assenmbler test programs, he ended up devising his own ingenious hardware add-ons. Currently, he is still training in Microsystems at the Dundee institute of Technology: his programming is done at night!

Although *Menace* was written entirely on the Amiga, Dave currently uses a PDS system running on a 386 PC with which to write. This system was used in the writing of *Blood Money*, the awesome follow-up to *Menace* released in May of 1989. Dave is a great fan of the Amiga and, as you will discover, certainly knows his onions from his hardware sprites…

Finally, *Amiga Format* would like to say thank you to all at DMA Design and at Psygnosis for their support and assistance with this feature series. Without whom it would not have been possible…


AMIGA Hardware Reference Manual
Commodore Business Machines, Inc.

Welcome to a series of articles in which most aspects of games programming will be discussed in depth. More specifically, and quite naturally, it will be aimed squarely at Amiga games programming. Games are made much simpler on the Amiga by the abundance of specific hardware that the machine possesses to handle the kinds of work games require.

I will assume some knowledge of 68000 programming. There have been many articles written on this subject, and good books available, for some time now. One book that is pretty essential is the bible of Amiga games programmers, the Hardware Reference Manual.

Source Secrets

To try to discuss game programming in general is a little difficult, because there is an unlimited variety of methods & tricks that are employed by different programmers. So, to give us a bit of direction, these articles will be accompanied by the full source code to an Amiga-specific game: namely my first game, *Menace*.

Source code to games is generally kept hidden away under lock and key, because it is the culmination of many months' work on the part of the programmers and a fair bit of the source code is usually carried on to other projects. It will be invaluable to this series, and hopefully beyond it, in getting across exactly how a game is designed & written.

Each month a specific part of the game will be documented, accompanied by the source code for that section. *Menace* should be of some interest as it does make use of a lot of Amiga-specific hardware: hardware sprites, dual playfield, hardware scroll, screen splits and so on (even though the game may look a little old these days!)

Defining our Terms

Some terms that are used in games programming may cause a little confusion, so first here is a short-list and description of the main ones used by programmers.

VERTICAL BLANK or FRAME – Essentially 1/50th of a second, the time it takes for a TV or monitor to update its display. An important factor for a game is the speed it runs at. The fastest will be 50 frames per second, ie the game runs as fast as the TV or Monitor can update. This leads to the silky-smooth scrolling of some games (like *Menace*, grin!) which can only be achieved at this speed. You can scroll slower, say 25 frames per second, but this starts to introduce a slight shimmer to the graphics. It may be a surprise to learn most 3D games only run at about 10 frames per second, which shows the scope for improvement if we had very fast hardware.

RASTER/SCAN LINES – Raster lines are basically the horizontal lines produced by the monitor which are related to the vertical resolution of an Amiga screen. Most games use 200 or more lines of display. NTSC displays used in the states can display a maximum of about 220 lines. PAL systems such as ours can display about 270 lines. The Amiga is a lot more flexible than other machines as it allows us to define our own screen sizes. The NTSC system is why so many games have a large black border at the bottom of the screen: what fills our screen by two thirds will give a full screen on an NTSC system. Not many programmers go to the trouble of producing two versions due to the large number of changes needed to the game (myself included) but full marks go the programmers who do (Dino Dini with *Kick Off*, for example).

TIMINGS – One method often used to judge how fast a piece of code is taking to execute (rather than adding up all of the instruction times: no mean feat!) is to change the background colour of the display to a certain colour at the start of the piece of code, then reset it back to the original colour at the end of the code. This gives a visual colour bar fidgeting about on the screen, which is a nice indication of roughly how many raster lines the code is taking. Next time somebody says 'I can clear the screen in about 100 raster lines' you will know what they mean.

DOUBLE BUFFERING – A technique that entails using two copies of the game screen. While one is being displayed the other is being altered, moving all the aliens about for example, this cuts out all forms of 'flickering' caused by changing a screen while we are looking at it. It is quite hungry on memory due to the two screens, but is fairly essential for smooth animation.

HARDWARE/SOFTWARE SPRITES – The Amiga has the facility of displaying hardware sprites which is a very fast way of putting objects on the screen. There is no visual way to tell the difference between hardware and software sprites: software ones are drawn into the actual screen memory. Hardware sprites are a little limited on the Amiga, but can be used for speed. The main ship in *Menace* is made up of hardware sprites, but all of the aliens are software sprites. Many people refer to software sprites on the Amiga as BOBS, short for Blitter OBjectS, as they tend to be drawn using the blitter.

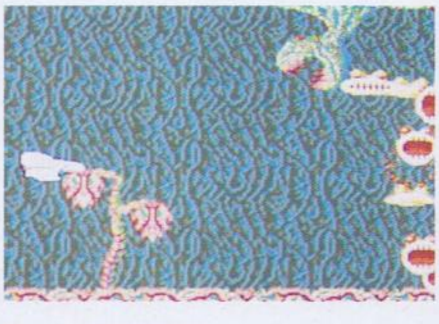
MASKING – When drawing graphics into the screen it is preferable to leave intact the graphics that are already there. This is done by masking, which lets all 'holes' in the graphic that we are drawing show the graphics underneath. The blitter in the Amiga is an expert at doing this for us.

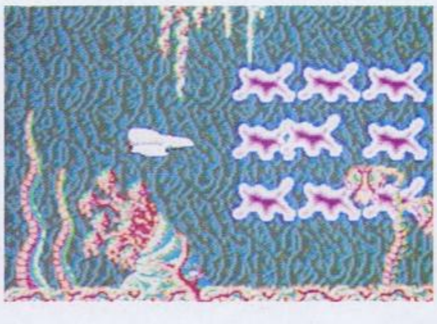
EDITOR – Not a text editor, but a piece of software that allows the editing of game data such as level maps, or alien movement patterns. These are quite time-consuming to write but save a lot of time once completed. Menace has no editors: it was the first game I had written, and all data was typed in by hand. Halfway through the game I thought "Boy, do I need an editor!" but never got round to writing one. Unless you really enjoy a lot of typing, one is strongly recommended. Even one written in another language like BASIC will suffice: but the best ones are usually integrated into the game allowing you to edit data at the press of a key.

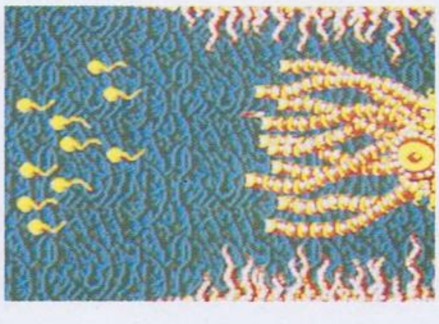
This Month's Source

The source file on the Coverdisk (framework.asm) is a small but invaluable program. Most games tend to 'bash the metal' which simply means that the operating ➧

THE MENACE WITHIN







system is not used – 'trashed' – which leaves us with 512K of free memory and full control over all of the hardware. This is required near the end of writing a game when memory may be short, but it means having to reset the machine and reload the assembler and source, each time we test a program.

To get around this when trying out programs we can be nice the operating system by properly allocating some memory, using DOS to load some files, then WHACK, hit it where it hurts and take over the system. Once our program has done what it wants we revive the operating system: it has no idea what happened, so it carries on as usual.

This allows us to test virtually every aspect of a game as if it had complete control of the machine. Of course if there are bugs in the code being tested which cause a crash, a reset will have to be performed. It is always nicer to work from RAM disk but be sure to save to disk regularly. A recoverable RAM disk is very useful if you have expansion memory. ASDG produce one (VDO:) which is by far the most bomb-proof: *Menace* was completely written using this, yet it survived 99% of crashes.

Framework uses the minimum of operating system routines to get by. This is the only time in this series that operating system routines will be used, so a quick run-through of their use is in order before we delve into the more meaty hardware.

OpenLibrary/CloseLibrary

To get access to certain system routines, such as DOS loading, requires us to open an associated library, which simply returns the address of a table containing some variables and addresses of the routines to call. Framework opens the graphics library to find the address of the system copperlist (more about this later). It also opens the DOS (Disk Operating System) library to access disk routines.

AllocMem/FreeMem

An exec library routine (the exec library is always in memory) to ask the system for some free memory is called. Even if you multitask your assembler there should be around 200K free for testing. Framework will simply exit if not enough memory could be allocated. Only CHIP memory (the specialist hardware can only access the first 512K, termed chip memory) is allocated because virtually all data used by a game has to be accessed by the hardware.

DOS Open/Read/Close

There are no DOS routines in framework at the moment as there was no need at this stage. These will appear next month to allow us to load any file into our allocated memory. Files can also be included straight into the source with the INCBIN directive: however, this tends to make assembly time quite long. DOS routines are simple to use so we'll take this path.

The above is the full extent of the operating system routines used. The rest of Framework basically consists of two routines, TakeSystem & FreeSystem.

TakeSystem saves all the vital information about the system, namely copper list addresses, and DMA and interrupt status. The system is then stopped by disabling all interrupts and DMA channels. This frees us to set up our own values.

Between the TakeSystem & FreeSystem calls is where our code will sit until FreeSystem is called, at which point the system is revived and we will be returned to the CLI.

If you run Framework as it stands just now, not a lot will happen. The screen will blank to the background colour, the mouse pointer will disappear and the usual disk drive clicking will vanish. The system is now dead, waiting for the left mouse button to be pressed. Press the mouse button and everything will return to normal.

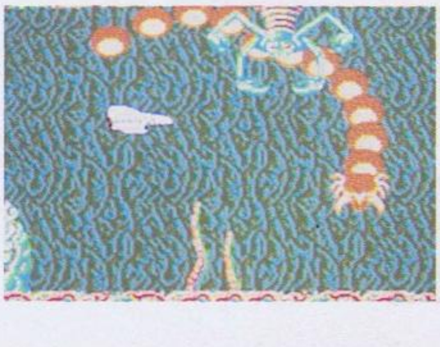
Note that we did not clear the screen in Framework, yet it did disappear. This is because we turned all DMA (Direct Memory Access) off. The Amiga uses DMA extensively when it requires to fetch or move memory. All the custom chips use this feature to fetch the data they need (blitter, sound, sprites etc) and we can selectively

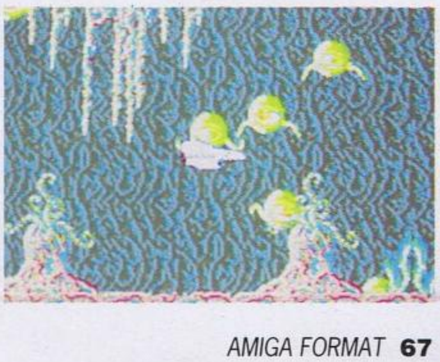
THAT MENACE SOURCE CODE...

Here is a complete listing of the source code included on this month's Coverdisk. Framework takes over and shuts down the Amiga system so that the game can do what it likes. You can type this listing in using a text editor if you so wish.

```
* Amiga system takeover framework
* 1988 Dave Jones, DMA Design

* Allows killing of system, allowing changing of all display &
* blitter hardware, restoring to normal after exiting
* Memory must still be properly allocated/deallocated upon
* entry/exit
* DOS routines for loading must be called BEFORE killing system

* Written using Devpac2

section Framework,code_c


* READ ME !!!
* The following block of conditional code is included to provide
* full compatibility with Argonaut's ArgAsm assembler system. The
* include files provided with ArgAsm are different from those on
* the Devpac program disk therefore several extra assignments have
* to be made for the code to successfully assemble under ArgAsm.
*                                                - Jason H.

    ifd     _ _ArgAsm

    incdir"Include:"
    include          exec/funcdef.i
_SysBase  equ        $04
    elseif
    incdir"include/"
    endc

* END OF CONDITIONAL BLOCK

    include          libraries/dos_lib.i
    include          exec/exec_lib.i
    include          hardware/custom.i

Hardware      equ    $dff000
MemNeeded     equ    32000
SystemCopper1 equ    $26
SystemCopper2 equ    $32
PortA         equ    $bfe001
ICRA          equ    $bfed01
LeftMouse     equ    6

start lea     GraphicsName(pc),a1  open graphics library purely
      move.l  _SysBase,a6          to find the system copper
      clr.l   d0
      jsr     _LVOOpenLibrary(a6)
      move.l  d0,GraphicsBase
      lea     DOSName(pc),a1       open the DOS library to allow
      clr.l   d0                   the loading of data before
      jsr     _LVOOpenLibrary(a6)  killing the system
      move.l  d0,DOSBase

      move.l  #MemNeeded,d0        properly allocate some chip
```

Continued on Page 68





turn on or off their ability to do so. DMA does tend to slow the processor down if it is being used extensively: however, this method of fetching/moving data is a lot faster and more efficient than using the processor to do the same job.

Main Game Loops

To give an idea of exactly what routines will be covered later, we will look at the 'main game loop' for Menace. All games should have a main game loop. Through the use of descriptive labels in your source this should show virtually every stage of the game as it is processed. Cue *Menace:*

```
MainLoop bsr        WaitLine223
         not.b      vcount(a5)
         beq        TwoBlanks
         bsr        Checkplayfield2
         bsr        Moveship
         bsr        CheckCollision
         bsr        EraseMissiles
         bsr        LevelsCode
         bsr        UpdateMissiles
         bsr        Drawforegrounds
         bsr        PrintScore
         bsr        CheckKeys
         bsr        CheckPath
         bra        MainLoop

TwoBlanks bsr       Checkplayfield1
         bsr        FlipBackground
         bsr        Moveship
         bsr        Restorebackgrounds
         bsr        ProcessAliens
         bsr        SaveAliens
         bsr        DrawAliens
         bra        Mainloop
```

As well as the above routines we will also need extra ones that are not used in the main game. These will be high score, initialise, text printing etc. Each routine should be as independent as possible from each other. By this I mean it should be possible to remove one of the above routines from the main loop, and still run the game: obviously with funny effects, but the game should not crash. This greatly helps when debugging a game as it nears completion.

Some of the most obscure bugs are when areas of memory may be being corrupted. With a main game loop constructed of individual routines we would successively remove individual routines until the bug vanished: this way we will at least know in which routine the bug lies. Well, at least 90% of the time!

Data Structures – the essence of a game.

Anybody who has taken courses in programming should have had the concept of data structures hammered home to them. Designing good data structures for your game data CANNOT be over emphasised. A data structure is simply a definition of exactly what data, and in what order, is needed to describe and control a certain object.

Take for example an alien moving about the screen waiting to be blasted. The information we need on this alien may be X & Y coordinates, number of frames of animation, where it is going, how may hits to kill it, how many hits has it taken, etc etc. To write code to move each alien individually would be very wasteful of time and memory, and be very inefficient. One or two routines should be written that control every alien by working on a data structure that is common to all aliens.

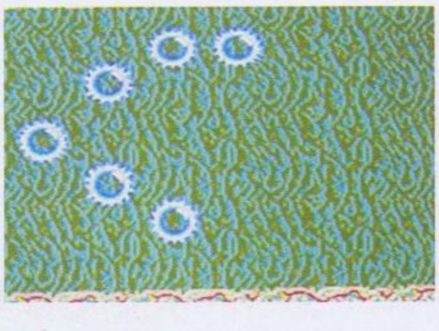
Most programmers tend to work this way as it is a fairly natural way to do things. Try not to cut down on what data your structures contain in the hope of saving memory. Complete game code, with all the data structures, tends to use about 10%-15% of the available memory, the rest being used for graphics, displays, sound etc. (other games, such as 3D ones, may differ). The ProcessAliens routine from the main game loop simply processes data structures, and nothing else. This will be described in full later.

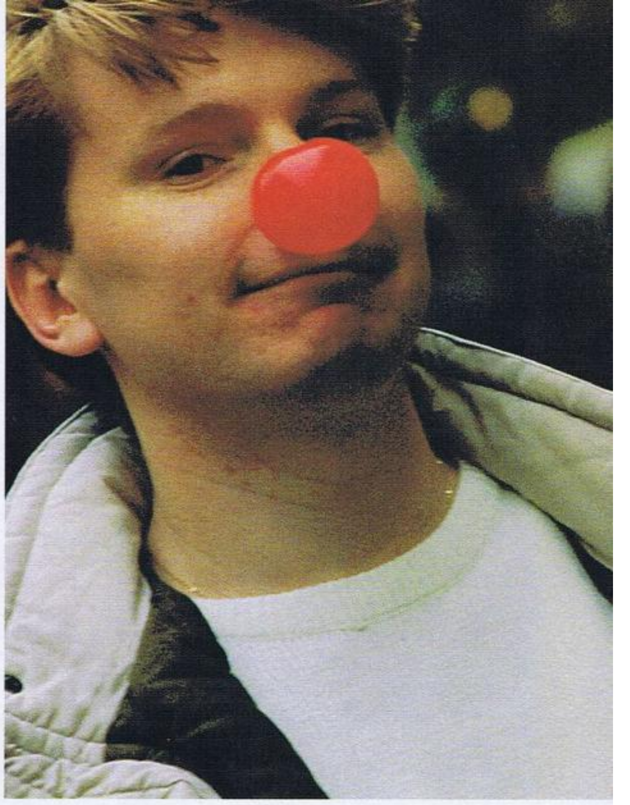
Next month will see the start of the really juicy programming bits with the source for the dual playfield scroll routine. ■

Continued from Page 67

```
      moveq.l #2,d1                  memory for screens etc.
      jsr     _LVOAllocMem(a6)       d1 = 2, specifies chip memory
      tst.l   d0                     where screens,samples etc
      beq     MemError               must be (bottom 512K)
      move.l  d0,MemBase

        move.l     #Hardware,a6        due to constant accessing
        bsr        TakeSystem          of the hardware registers
*                                      it is better to offset
wait    btst       #LeftMouse,PortA    them from a register for
        bne        wait                speed & memory saving(A6)

          bsr     FreeSystem

          move.l  _SysBase,a6
          move.l  MemBase,a1
          move.l  #MemNeeded,d0          free memory we took
          jsr     _LVOFreeMem(a6)
MemError  move.l  GraphicsBase,a1
          jsr     _LVOCloseLibrary(a6)
          move.l  DOSBase,a1             finally close the
          jsr     _LVOCloseLibrary(a6)   libraries
          clr.l   d0
          rts

TakeSystem  move.w  intenar(a6),SystemInts save system interupts
            move.w  dmaconr(a6),SystemDMA  and DMA settings
            move.w  #$7fff,intena(a6)      kill everything!
            move.w  #$7fff,dmacon(a6)
            move.b  #%01111111,ICRA      _ kill keyboard
            move.l  $68,Level2Vector       save interrupt vectors
            move.l  $6c,Level3Vector       as we will use our own
            rts                            keybd & vblank
* routines

FreeSystem  move.l  Level2Vector,$68       restore system vectors
            move.l  Level3Vector,$6c       and interrupts and DMA
    move.l  GraphicsBase,a1                and replace the system
    move.l  SystemCopper1(a1),Hardware+cop1lc     copper list
    move.l  SystemCopper2(a1),Hardware+cop2lc
    move.w  SystemInts,d0
    or.w    #$c000,d0
    move.w  d0,intena(a6)
    move.w  SystemDMA,d0
    or.w    #$8100,d0
    move.w  d0,dmacon(a6)
    move.b  #%10011011,ICRA                keyboard etc back on
    rts

Level2Vector dc.l   0
Level3Vector dc.l   0
SystemInts   dc.w   0
SystemDMA    dc.w   0
MemBase      dc.l   0
DOSBase      dc.l   0
GraphicsBase dc.l   0
crap         dc.b   0

    even
GraphicsName dc.b    'graphics.library',0
    even
DOSName      dc.b    'dos.library',0
    end
```

Note that the tabulation and the 'comment' asterisks may vary.





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).

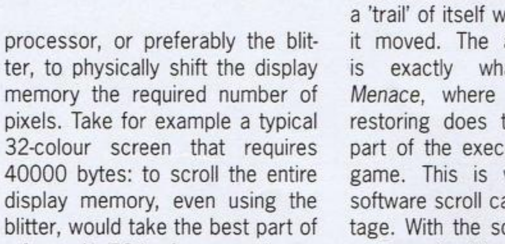
Pros and Cons

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.



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.

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 ▶

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

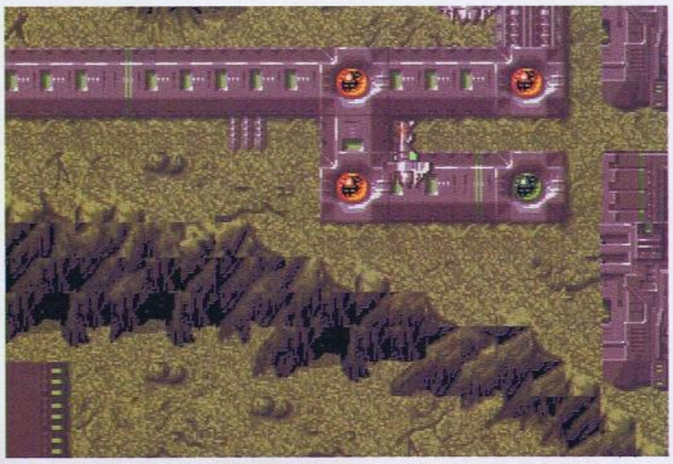
The 192 line height of the playing area was chosen as it is the closest multiple of 16 to 200, the game panel adds another 32 pixels to the overall height bringing the full screen size to 224 pixels. The background playfield is constructed in a similar way (see figure 2) but requires an extra 32 pixels 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.**

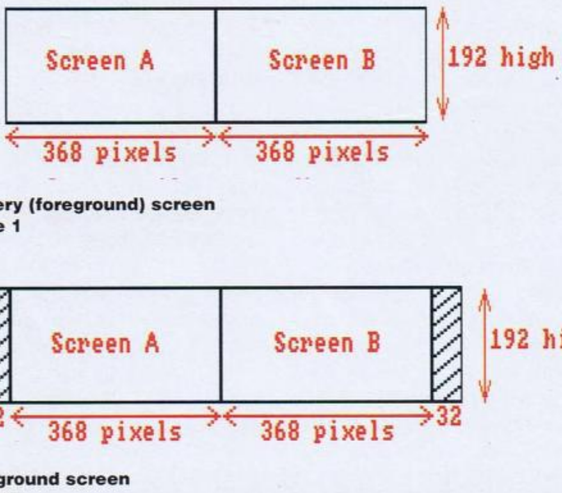
How the scroll works...

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.

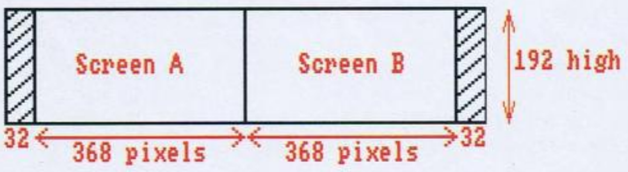
Take a look at figure 3. This shows our two screens laid side by side in memory. At any one time we are displaying 352 pixels (22 words) of this data. The bit plane pointers on the Amiga can be positioned on any word (16 pixel) boundary. Incrementing the pointers therefore would scroll through memory 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.

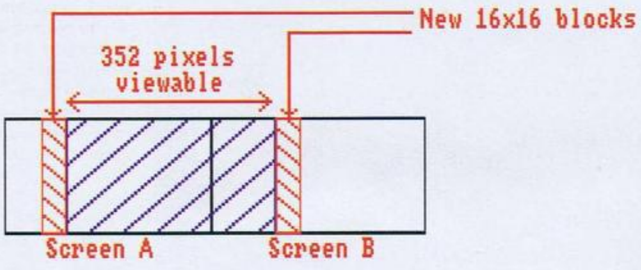
Right, if you understand so far you may notice a quirk in that when we have fully reached screen B, we reposition the plane pointers back to screen A and start again. This sudden jump to screen A though will cause a complete new screen to appear showing what was previously in screen A. This is where 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



Scenery (foreground) screen
Figure 1



Background screen
Figure2



Scenery (foreground) screen
Figure 3

***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.

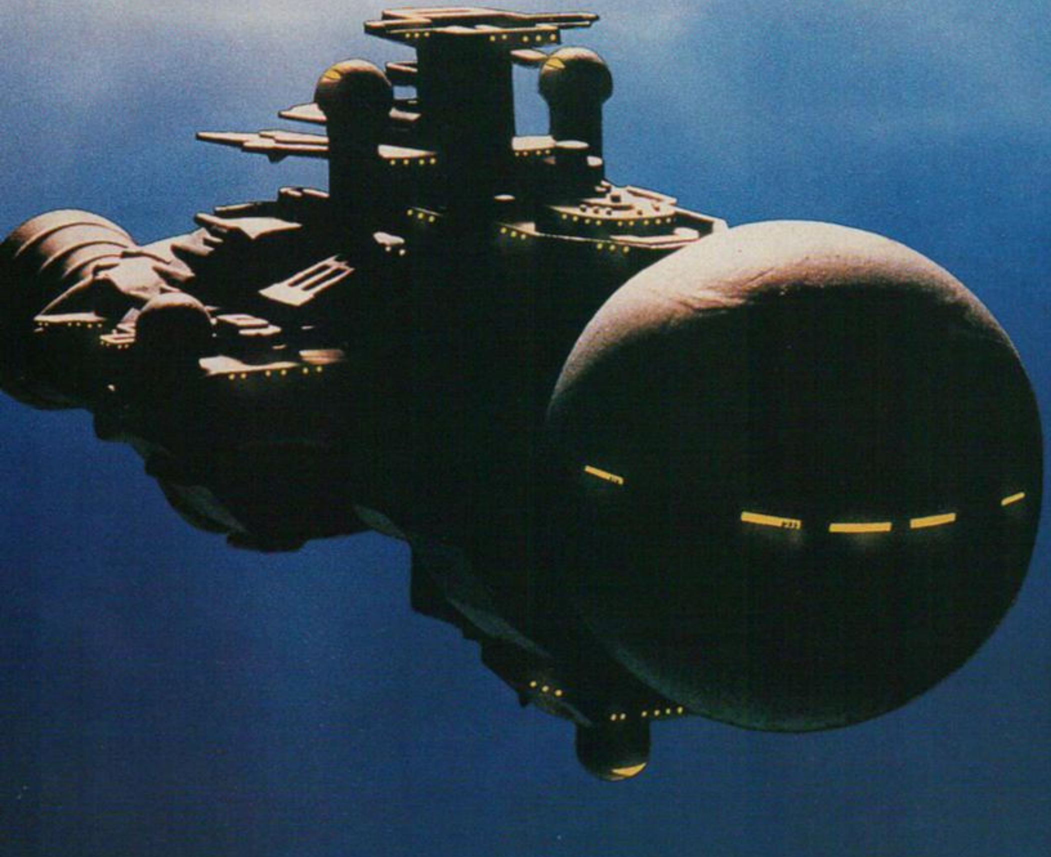
COLOR and SPRITE registers are all set to 0 initially, these are set up by the initialisation routine of the game.

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.valueDC.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
rastersplit2DC.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
```

DESIGNING THE MAIN SHIP



THE WHOLE TRUTH ABOUT GAMES PROGRAMMING: 3

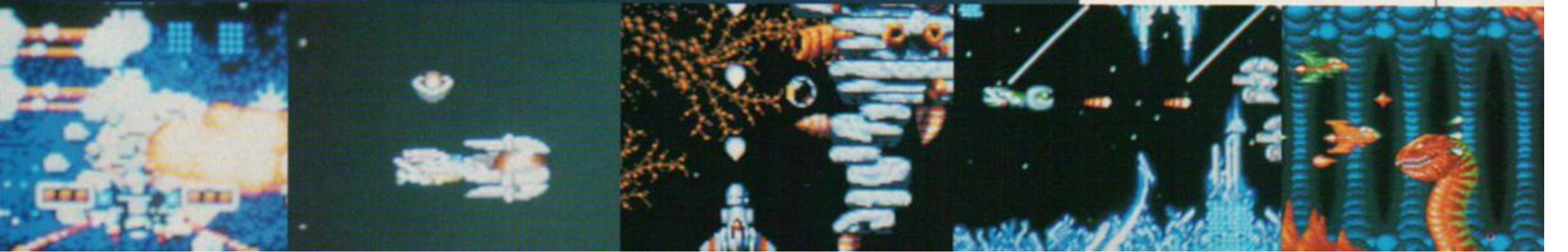
Top programmer **DAVE JONES**, author of Psygnosis' hits *Menace* and *Blood Money*, reveals more secrets of the art. This month:

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.

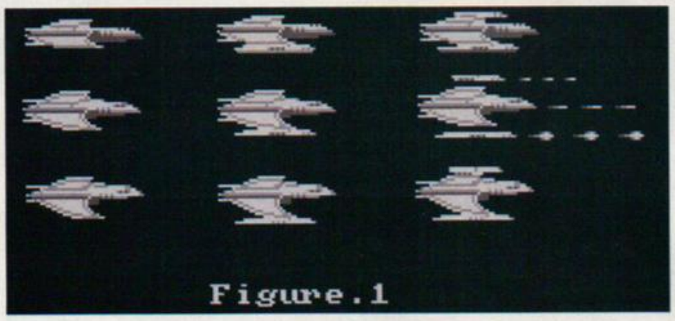
The inertia is simply a snippet of code that prevents you instantly switching direction, and instead forces the ship to slow down in the direction it was going, stop, and then accelerate to its maximum speed in the chosen direction. It is not so noticeable on the initial speed of your ship, two pixels 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

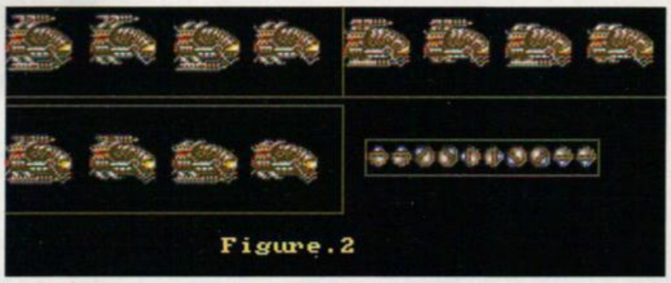
ship animation = (184+96)*2 (due to overlaying) = 560 bytes ➤

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.



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.

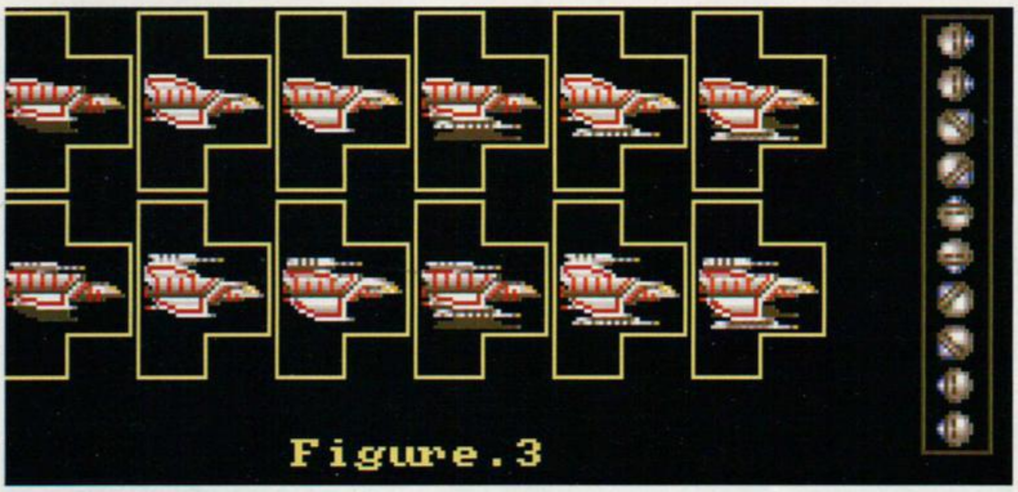
Tricks and Treats

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.



THE WHOLE TRUTH ABOUT games programming PART 4

aliens

DAVE JONES, the ace coder behind Psygnosis' hits *Menace* and *Blood Money*, spills more secrets.

In this and next month's articles I will detail the largest part of the *Menace* code: namely, the alien movement and control routines. There will be no source with this month's instalment because I will only be describing WHAT the routines accomplish – next month will cover HOW they are accomplished, along with the source.

Movement is Life

So, we want to have aliens flying about the screen in nice patterns, attacking your ship, launching missiles at you, tracking your movements and doing all the other things that aliens do best. All the basic ingredients of a good shoot-em-up. What should spring to mind to control all of the movements is a decent data structure. As I have said before, there is no substitute for sitting down with a pen and paper and having a good think about what has to be accomplished. The best idea is to start off with the basic data that would have to be stored about an alien flying about on screen. This would probably be the following:

Alien number
Alien X coordinate
Alien Y coordinate
Alien speed

There will also have to be some way of telling the alien what to do. The simplest form would be a coordinate for it to move to. Once it gets there it would need another coordinate to go to. It would do this repeatedly until we tell it to stop, or until is has been destroyed. First, then, we will define the basic structure to describe a particular alien

```
            rsreset        ; reset the rs counter
X.Pos       rs.w     1     ; one word for the alien's x coord
Y.Pos       rs.w     1     ; one word for the alien's y coord
Sprite.Num  rs.b     1     ; a byte to hold the alien number
Speed       rs.b     1     ; a byte to hold the alien speed
```

Following this in memory would be a table of x,y coordinates for the alien tc move to in sequence. An example piece of source that moves an alien from left to right and back again could be defined as follows:

```
dc.w=  0,100                     ; start alien at x=0, y=100
dc.b   6,2                       ; use alien number 6 with a speed of 2

dc.w   320,100,0,100,$7fff       ; move to 320,100 then to 0,100
                                 ; the $7fff is an end marker
```

This basic structure will provide only limited control of an alien, so it's time to add a few more features. Most of the aliens are animated (Figure One, overleaf, shows an eight-animation *Menace* alien) so as well as defining the alien number, we also have to define its current animation number. Note that in games an animated object is usually given a number that relates simply to the base address of that alien. The number of animations it actually has is stored separately. This makes changing the number of animation frames very simple, because the alien numbers all remain exactly the same. So we will also have to store the number of animations the alien contains somewhere in the structure as well as its current animation number:

```
Anim.Num    rs.b   1       ; a byte to hold the current animation
Num.Anims   rs.b   1       ; a byte to hold the maximum anims
```

The basic idea, then, is that for each game cycle we increment the

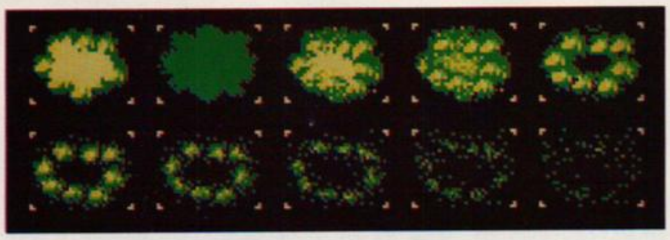


FIGURE ONE: AN EIGHT-ANIMATION *MENACE* ALIEN

Anim.Num (0,1,2,3,... etc) until it hits the value in Num_Anims when we would reset it back to zero. Straight away a small problem crops up in that we may not want to animate an alien every game cycle (1/25th of a second for *Menace*). In fact very few of the aliens animate every game cycle as this is a little fast, so we will also have to introduce an animation delay into the structure:

```
Anim_Delay          rs.b 1      ; a byte to hold the animation delay
```

The method of incrementing the Anin.Num until it hits a maximum then resetting it to zero is termed a WRAP animation as it wraps around to zero. In some cases we may want the animation numbers to increase to a maximum and then decrease to zero: this is termed an UPDOWN animation. To indicate whether or not to use a wrap or updown type of animation requires us to store a single bit of information. There are many occasions in the controlling of an alien that only requires a simple on/off definition. These tend to be grouped together in the data structure and called a MODE word or byte. This is simply a collection of bits used for general-purpose description of simple on/off controls. This is now added to the structure:

```
Mode        rs.b 1          ; single byte to hold eight flags

UpDownequ   1<<3            ; define the updown flag as bit 3
```

Note the equate of which bit in the control byte reflects the animation type. NEVER use 'magic numbers' in your source such as the statement

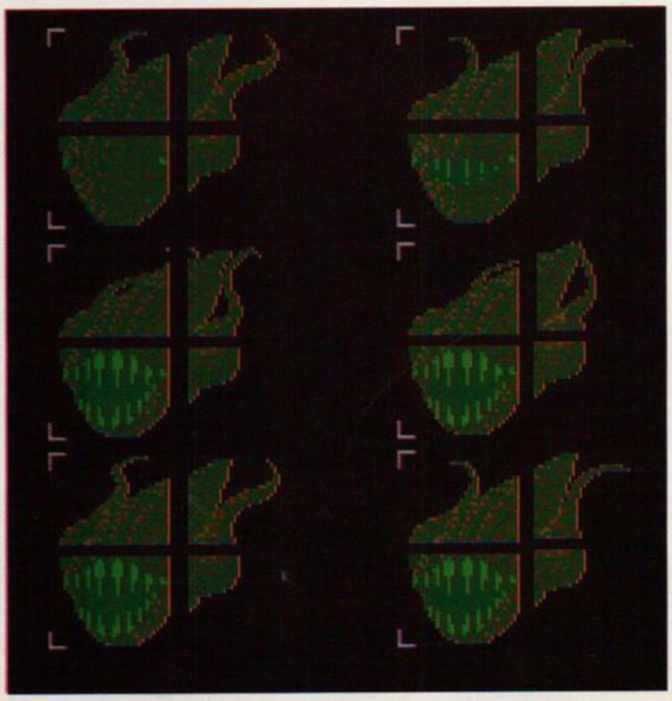
```
    btst  #3,Mode
```

This in no way tells you what bit 3 signifies: you may remember as you are working on the project, but what about in a year's time when you come back to re-use some of this source? The above should have been written as

```
    btst  #UpDown,Mode
```

which is very clear and has the advantage that to change the actual bit number that the UpDown flag is set at requires only the EQU statement above to be changed. Always make the assembler do as much work for you as possible. Right, lecture mode OFF, let's introduce the next element into the structure.

FIGURE TWO: FOUR SPRITES MAKE ONE LARGE SPRITE



Reason for Living is Dying

Most aliens' sole purpose for existing in a shoot-em-up is to be blasted. Do we want them to die after being hit once, though? For most, I think not. We will have to introduce a variable to define how many times we can fill an alien with laser shot before it explodes:

```
Hits_Num rs.b       1           ; number of shots an alien can take
                                ; $ff for infinite
```

As you can see a certain value has been introduced that makes some aliens indestructible. This is not used very often but is handy, say for some asteroids tumbling across the screen that you simply have to dodge and cannot destroy (anybody who reached Level 6 of *Menace* will know exactly what I mean!)

As far as describing the alien, that is about it. I have made no mention of the size of the alien in the structure as in *Menace* all aliens were 32 x 24 pixels in size. To make larger ones simply meant moving a few about as though they were joined together (see Figure Two). It certainly makes life simpler handling only one size of alien, though you do lose out a little on flexibility.

Back to Motion

Now back to the movement of the aliens. The basic GOTO command was implemented simply by defining pairs of x,y coordinates for the aliens to head for – they continued until they were destroyed or until a special coordinate value told them to stop.

This is very inflexible for performing, say, a small circle movement. We would have to define goto points for many points on the circumference of the circle. This may take quite a while to work out, and would have to be recalculated for any other aliens at different x,y coordinates on the screen that wanted to perform a similiar circle.

One simple way of overcoming this problem is to use a relative coordinate system whereby rather than interpreting the x,y values as goto coordinates, they are simply offsets that are added to the present x,y coordinate. This allows us to define a shape as a set of offsets from a base point, the base point being arbitary. This should be able to be used in conjuction with the goto points so we can switch freely between absolute and relative movement at any time.

It is apparent that some form of control data has to be added into the coordinate list to switch between relative and absolute or one may be mistaken for the other. What basically happens in *Menace* is that movement always starts off in absolute mode, but at any time we can switch to and from relative mode by making one of the coordinate values a control word. For example to goto the middle of the screen, then move in a triangle shape, then return again may look something like:

```
    dc.w  160,100           ; goto 160,100
    dc.w  OffsetMode        ; switch to offset mode
    dc.w  2,2               ; move 2 along and 2 down a few times
    dc.w  2,2
    dc.w  2,2
    dc.w  -2,0              ; move 2 back a few time
    dc.w  -2,0
    dc.w  -2,0
    dc.w  2,-2              ; move 2 along and 2 up a few times
    dc.w  2,-2
    dc.w  2,-2
    dc.w  OffsetMode        ; toggle back to absolute mode
    dc.w  0,100             ; back to the start x,y
```

Control words have certain bits set that distinguish them from a goto coordinate (not much point in, say, having 32768 as a coordinate with a screen width of 352 pixels). What we end up with is a small language that describes the path aliens follow. We can go on adding control features that make the path data more compact yet more powerful. The offset mode will set a bit in the MODE byte to indicate when it is in offset mode.

Memory Economy

To cut down the amount of data required to describe a complex path (like a large ellipse formed from offsets) it would be ideal to be able to branch to some other path data, and then return after executing so many of its commands. The triangle path above, for instance, requires 40 bytes for the offset data. If we required 12 aliens to perform the triangle movement, but at different coordinates on the screen, we only want to define the offset data once, and have the rest of the aliens branch off to that data inbetween executing their own path data. Menace has a GOSUB command in the path data for this purpose. The gosub command passes over to the new offset data, then at the end of the data is the equivalent of a RETURN command which returns control back to original path data. The above could be written as:

```
        dc.w    160,100             ; goto 160,100
        dc.w    Gosub,Newpath-*     ; branch to some new data
        dc.w    0,100               ; back to the start x,y


Newdata dc.w    OffsetMode  ; switch to offset mode
        dc.w    2,2         ; move 2 along and 2 down a few times
        dc.w    2,2
        dc.w    2,2
        dc.w    -2,0        ; move 2 back a few times
        dc.w    -2,0
        dc.w    -2,0
        dc.w    2,-2        ; move 2 along and 2 up a few times
        dc.w    2,-2
        dc.w    2,-2
        dc.w    OffsetMode  ; toggle back to absolute mode
        dc.w    Return
```

If we had 12 similar paths to the above the memory required would now be drastically reduced.

More Memory Economy

Another useful memory saver is a FOR type contruct that allows any part of the path data to be repeated a set number of times. Imagine trying to move a series of aliens in a 'stepping' motion whereby they move, say, along 16 pixels then down 16 pixels, achieved by:

```
    dc.w OffsetMode
    dc.w 4,0
    dc.w 4,0
    dc.w 4,0
    dc.w 4,0
    dc.w 0,4
    dc.w 0,4
    dc.w 0,4
    dc.w 0,4
```

The above performs one step, but if we wanted to repeat this 10 times, it would normally mean repeating the data 10 times or using many GOTO statements – but, yes you've guessed it, a loop function was implemented: simply appending the command

```
    dc.w DoLoop
```

to the end of the data will cause the last xx bytes of path data to be repeated so many times. This caused two new items to be added to the alien structure, namely

```
Loop_Offset     rs.b 1    ; the number of bytes to loop back
Loop_Count      rs.b 1    ; how many loops to perform
```

This limits the loop command to only allowing one size of loop in each individual path, although this restriction was not noticed as multiple loops were never required.

Other Tweaks

A PAUSE function was added that allowed an alien to pause at a specific point for any length of time. A simple function but used quite often:

```
    dc.w  Pause,PauseValue ; the Pause value is in 1/25 seconds
```

A SEEK mode was added that allowed an alien, at any time, to start to track your ship and try to collide with it to reduce your shield. This is worked similarly to the pause function with a count specifying how long to seek your ship before carrying on with the rest of the path data:

```
    dc.w  Seek,SeekCount      ; seek ship for 1/25 * SeekCount secs.
```

Related to the seek command above is the FIRESEEK command which allowed one alien to start another path. This was used to allow aliens to fire missiles that used the seek mode:

```
    dc.w  FireSeek,SeekCount ; start a seek path from this alien
```

To finish off the command set some additional simple commands that came in useful were added. These are:

ChangeSprite – allowed you to change the sprite number at any time. Useful for transforming aliens into other types.

ChangeSpeed – quite obvious this one. Slow moving sprites can be given a quick turbo boost to attack the ship.

ChangeAnim – allows manual changing of the animation number. Used to mimick, say, turning a corner at any point on the screen

And that, basically, is that. Table 1, below, lists the full data structure for an alien path. Hopefully you can see the benefit of designing such a control system for moving objects about. It need not necessarily only be used in a shoot-em-up style game but can relate to many styles of game. It soon becomes quite simple to add more powerful commands to the code, which results in the code being useful in many more projects.

Now you know what the commands do – next month I will present the source and explain how they are accomplished. ■

TABLE 1
FULL DATA STRUCTURE

rsreset			
Next.Path	RS.W	1	offset to the next path
X.Pos	RS.W	1	current x position
Y.Pos	RS.W	1	current y position
Kills.What	RS.W	1	kills others aliens if dead (0-11)
Table.Offset	RS.W	1	the current table offset
Sprite.Num	RS.B	1	sprite number
Anim.Num	RS.B	1	animation number
Anim.Delay	RS.B	1	delay in 1/25th secs, dynamic copy
Anim.Delay2	RS.B	1	static copy to refresh the above
Speed	RS.B	1	speed in pixels
Pause.Count	RS.B	1	dynamic pause counter
Mode	RS.B	1	flags, see bleow
Loop.Offset	RS.B	1	loop offset in bytes (-ve)
Loop.Count	RS.B	1	dynamic loop count
Hits.Num	RS.B	1	number of hits to kill
Num.Anims	RS.B	1	number of animations
Seek.Count	RS.B	1	dynamic seek count
Table.Size	RS.B	0	

* This is followed by x,y bytes to move to (always even) with command

* codes inserted to alter the control and movement.

*	Mode bits			
	Offset.Mode	equ	1<<0	set when in offset mode
	Seek.Mode	equ	1<<1	set when in seek mode
	UpDown	equ	1<<3	set when updown animation
	AnimUpDown	equ	1<<4	set to animate up, reset down
	HeatSeekPath	equ	1<<5	set to signify a heatseeker

THE WHOLE TRUTH ABOUT games programming PART 5

aliens 2

As we get close to the complete *Menace*, **DAVE JONES** fills in the details of how to animate aliens.

Following straight on from last month's ramblings about the features required by the alien movement routine, this month's Coverdisk contains the source code that implements all the functions discussed last time. It is quite lengthy – lucky you dont have to type it in! – and consists of three main sections:

1. The control of aliens in a path.
2. The starting/stopping of paths.
3. Drawing the aliens in each path.

The drawing of the aliens is broken down into three further stages:

1. The replacing of backgrounds saved from the previous printing of the aliens.
2. The saving of the backgrounds where the next set of aliens is to be drawn.
3. The actual drawing of the next set of aliens.

Big Clipper

Note that no clipping of BOBs (Blitter OBjects) is carried out in *Menace*. Clipping, a very common feature in games, ensures objects move smoothly onto the screen from the borders rather than just instantly appearing. The aliens in *Menace* do not appear instantly, though, so you may have realised that some form of clipping must be taking place.

The simplest way to achieve a clipping effect is to make the physical screen size larger than the one that is being displayed. In *Menace* all aliens are a maximum size of 32x24 pixels. If this area is added to each side of the displayed screen size it gives us an area of the screen into which we can draw an alien that will not be displayed (see Figure Two). Once we start moving the alien onto the screen it will glide smoothly on.

Once again, the trade-off between speed and memory comes into play. We can keep the screen the same size as the displayed one and use software to calculate how much of the alien is clipped, only drawing the correct amount, or we can sacrifice the extra memory to dispense with the software clipping. In some games it becomes essential to use a software clip. Basically if you plan on having large moving objects in a game, then there will probably not be enough memory to allow the extra screen size around the displyed screen to accomodate and hide large objects.

Blitter Pill

The aliens are drawn using the blitter (surprise, surprise). The blitter is much, much faster than using the 68000 to move memory around – even small bits of memory – and let nobody try to tell you different, especially ST owners. Blitter sprites are masked, shifted and drawn in one operation for each plane (three planes in all).

The graphic data is stored in the most common way for blitter data. This is each plane of graphic data stored sequentially in memory, with a plane of mask data last. The mask is simply all the planes of data ORed together. The mask is used to 'cut' out of the screen the pixels where some data from the BOB has to be placed. Without this, the pixels that are there affect the BOB data resulting in the wrong colours appearing.

Alien stored as:

Plane 1 4 bytes x 24 scanlines = 96 bytes

Plane 2 4 bytes x 24 scanlines = 96 bytes

Plane 3 4 bytes x 24 scanlines = 96 bytes

Mask 4 bytes x 24 scanlines = 96 bytes

Total = 384 bytes per anim

To draw an alien BOB requires three separate blits, one for each plane of the screen we are drawing into. All four blitter channels are used and are assigned to:

Blitter A channel = mask
Blitter B channel = data
Blitter C channel = screen
Blitter D channel = screen

Two channels point to the screen data as the screen is used as both a SOURCE and a DESTINATION channel. The blitter is used to perform the function of ANDing a

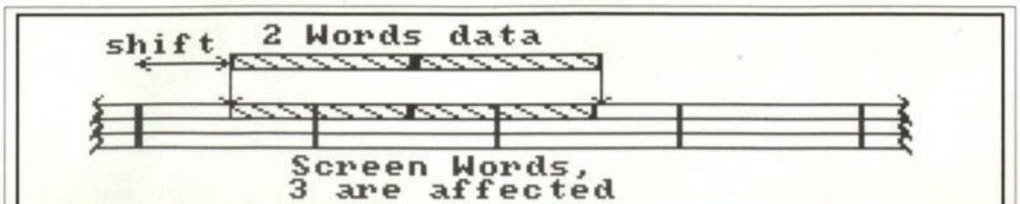


Figure 1

word from the screen with the inverse of the mask word and then ORing the data into the result. This function first removes the data from the screen where the mask has >1 bits present, then draws the data into these bits. This is performed for all three planes. The one mask is applied to all planes but the data for each plane is, of course, different.

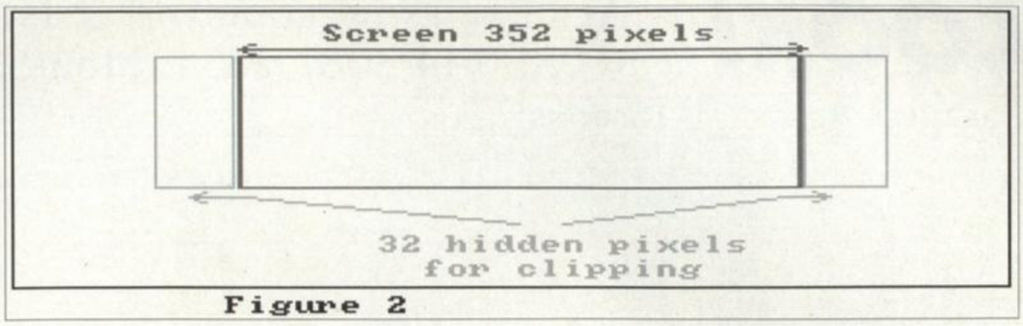


Figure 2

Shift Work

The shifting operation incorporates a neat little solution to a problem that many people have asked me how to get around. If a 32-pixel wide BOB is to be drawn onto the screen with the blitter, then we would tell the blitter it is two words wide by however deep. This is ONLY true if the BOB is not shifted: ie its X position is a multiple of 16 pixels (or an even number of bytes).

As soon as we want to place a BOB on ANY pixel position we will be affecting THREE words in length on the screen (see Figure One). This is due to the fact that we must use the blitter to shift the BOB right as it draws it into the screen. The common solution most people use is to store their graphics with an extra word at the end of each scanline: ie aliens in *Menace* would be stored as 48x24 pixels in size. This extra word is for the blitter to shift the data into: a maximum shift of 15 is needed.

Note that the the blitter width when drawing shifted BOBs is always one word greater than the actual BOB width to accommodate for the shift. This overhead obviously increases the memory required to store all the BOBs you may have to store (unless they do not require to be shifted as in character sets). There is indeed a way the blitter can handle this, although not documented in the hardware reference manual.

The basic aim of the problem is to give the blitter an extra word of zero at the end of each scanline, but still manage to store the BOB at only 32 pixels wide. Now, if we do store the BOB only 32 pixels wide but blit on the 48 pixels required what will happen is that some extra data will appear on each scanline where the BOB is drawn. This will be data from the next scanline down in the BOB due to the blitter fetching this extra word. In effect, the LAST WORD of the scanline will contain data and not zeroes.

However, the LAST WORD should ring a bell when we are talking about the blitter as the blitter has a feature called first and last word masks for its A channel. Normally these each have the value $FFFF, which masks off no data in the blit (the first word in the A channel blit is ANDed with the blitter A first word mask, and similiarly the last word in the A channel blit is ANDed with the blitter A last word mask). If, however, we set the last word mask for channel A to $0000, and assign channel A as our mask channel (ie channel A will point to the mask for each plane blit) then the extra word of data picked up by the blitter will be ANDed with the word $0000 which will force the data to be zero. This then gives us the desired effect of having a zero word at the end of each scanline of data.

One small side effect still remains, though, in that the blitter has fetched an extra word of data for the BOB, this data coming from the next scanline down in the BOB data. We therefore have to change the MODULO for the mask & data channel to a value of –2. The modulo is simply a value added to the blitter pointer at the end of each scanline. This would normally be zero if the data was arranged sequentially, but because we have fetched an extra word of data, we have to pull back the blitter pointer to that extra word otherwise every subsequent line of data would be out by 2,4,6... bytes of data. So to summarise:

1. Data is stored sequentially as Plane 1, Plane 2, Plane 3, Mask
2. The exact size (4 bytes x 24 scanlines) only is stored
3. Blitter channel A will point to the mask for each blit
4. Blitter channel B will point in turn to each plane of data
5. Blitter channels C & D will point in turn to each screen plane

The width of the blit will be 3 words, as the data is only 2 words wide the last word will be masked to 0, and the modulo will be –2 for the mask and data channel

The modulo for the channels C and D is the width of the screen minus the width of the blit.

Having a bliter to handle most of the drawing of objects is really a godsend on the Amiga. Try switching to a machine that has no hardware support for drawing, where you rely only on the processor, and you are immediately faced with many problems and compromises in trying to achieve what the blitter can handle.

This is one of the reasons why Amiga games can be exactly the same as an ST version if it was developed first. Porting a game from the ST to the Amiga can usually be done in a matter of days with no problems. Take a game written for the Amiga, though, that makes heavy use of the blitter, and you will need some major rewriting of code. This is the main reason why people tend not to make full use of the Amiga hardware when designing and writing a game for both 16-bit machines.

Back on the Path

Now back to the path movement control that was discussed last month. The final bit to explain was how the commands are actually defined as data. The file 'paths.s' on this month's Coverdisk contains the data for all the paths for Level One of *Menace*. All the paths were designed and entered by hand. This is not the ideal way to do things, some form of path editor would have been better, but if it's your first game you are writing you tend not to be too ambitious. It's best to concentrate on actually finishing a game!

A single path starts with the definitation data exactly as described in last month's structure. This describes the speed, animations, etc.

Following this is the movement data. We basically had two types of data:

1. A coordinate pair which were relative or absolute.
2. A command byte with optional parameters.

Looking at the coordinates first, we must decide upon the maximum values these can be. For relative coordinates a limit of +/- 16 would suffice. For absolute coordinates we have to look at the screen size to determine the limits. Basically the minimum X & Y will be 0,0 . The screen is 352 pixels wide, but 32 pixels are added to the left and right for the clipping as described. This gives us a maximum X,Y of 384,168 (note the x,y coordinate defines the upper left of the BOB, so although the screen is 192 pixels high the maximum Y is 192-24 = 168).

The BOBs are not clipped at all on the Y coordinate as they appear behind the foreground on the dual playfield screen. Thus the border graphics that are always present top and bottom hide the fact that BOBs can suddenly appear at the top or bottom. From the maximum values we see that the Y coordinate would fit in a byte but the X coordinate would need a word to hold any value. To save memory and keep both coordinates the same it was wise to store the absolute coordinates divided by two. This imposes a slight restriction in that only even coordinates are allowed, but this is never noticeable. We can now store the x,y absolute or relative coordinates in a byte with a value from –16 ($f0) to 192 ($c0).

Illegalities

As there are some values that are illegal when it comes to the coordinates ($c1 to $ef) we can embed the control commands discussed last month into the coordinates to further save memory. A maximum of 16 commands were allowed and these were assigned the values $e0 to $ef. The flow was then along the lines of:

```
    Fetch the X,Y coordinate bytes
    If X bytes is in the range $E0
– $EF then execute a control
    command
    else if the offset mode bit is
set the coordinates are
    relative and are added to X,Y
    else the coordinates should be
advanced towards.
```

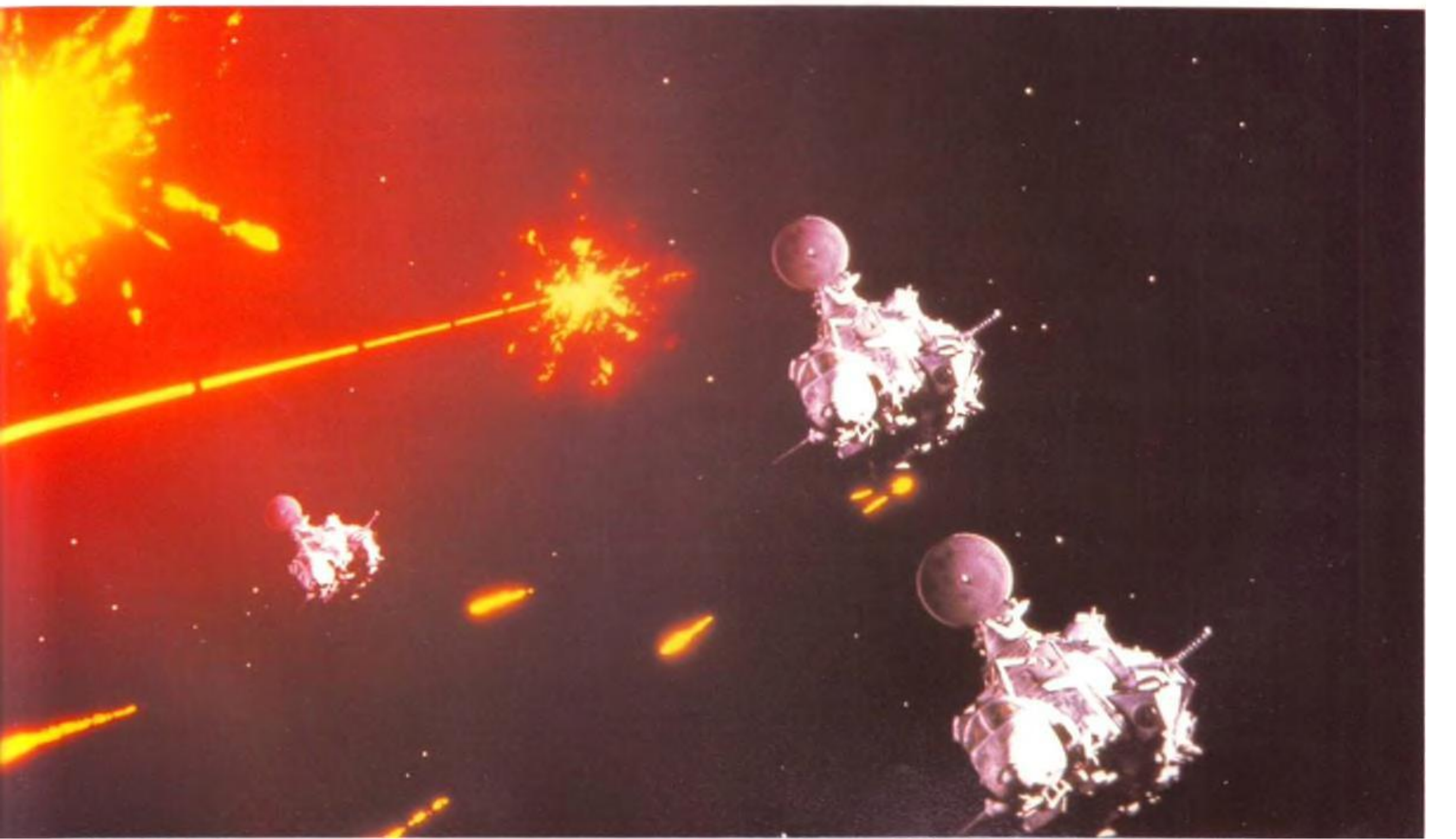
And that is basically that. This month's source implements this form of control, the data being acted upon is in the file 'paths.s'. It is very simple to try changing the coordinates and commands to form your own paths. ■

Next month will see the game becoming playable with the firing and collision detection routines added so at last you can blast and be blasted!

THE WHOLE TRUTH ABOUT games programming PART 5

Collision detection

The *Menace* code is nearly complete, becoming playable as **DAVE JONES** adds the bump-into-things routines.



This month sees all of the game coming together – and now being playable! – with the inclusion of the collision detection routines.

There is normally some form of collision detection performed in all games – one object moving into another object is a very common occurence that normally has to be detected. *Menace* required the following collisions to be identified:

- Player ship to aliens
- Player ship to foreground (expert mode and guardian)
- Weapons to aliens

These three are really all that is required in *Menace*. The first two of the above are very simple thanks to the Amiga hardware and the way *Menace* was designed.

Remember that the player's ship is a hardware sprite. The Amiga has the ability to detect collisions between any of the harwdare sprites and any bit plane(s) we specify. In the dual playfield mode that *Menace* runs in, bit planes 1, 3, & 5 form the back playfield, with the even planes (2,4 & 6) forming the front playfield. The aliens are drawn into the back playfield (planes 1,3 & 5) so we want the hardware to detect when the hardware sprites that form the ship collide with pixels in these planes.

A problem springs up here, though, in that the actual background graphics would register a collision as these completely fill up planes 1 and 3 – being four-colour, they only require two planes. Plane 5 is, however, untouched by the background graphics and will ONLY contain data from the actual aliens. So rather than detect a hardware sprite collision between planes 1, 3 & 5 we will only detect a collision to plane 5.

You may notice a small quirk here in that only checking the third plane in the alien graphic data will not be very effective if the alien used very few colours that have bits in the third plane set. In reality all of the aliens in *Menace* used mainly the colours that are included in the third plane set (colours 4, 5, 6, & 7) as these were redefinable for each alien. The background colours (0,1,2 & 3) were very rarely used in the aliens themselves.

Hardware Registering

The hardware register that controls the sprite-to-playfield detection is called CLXCON. This lets you select which bit planes to detect, and the value to detect (normally 1 for a set pixel). All that happens is that a mask of the ship is ANDed with the bit plane data. This will only register a collision when two bits overlap, which is indicated in the CLXDAT register.

This is simply a hardware implementation of a commonly-used software routine where an object's mask is ANDed with a screen mask to check for a collision. This type of detection is termed 'pixel perfect' detection as just one pixel of your ship hitting a single pixel of an alien will be detected. Many games use other methods of detection that are sometimes a little crude. How many times have you heard the shout 'No way I was dead, I was miles away from that bullet, stupid ❄◆✱✱❄■✢ game!'.

Ship to Foreground

The ship to foreground collision detection was handled simiarly, with a collision between plane 6 and the ship being registered. Initially I registered collisions between planes 2, 4 & 6, all of the foreground planes. This then exhibited a quirk because the ship missiles are drawn into the foreground playfield. If the ship had quite a few speed-ups you could fly over some of your own slower-moving missiles – and then you lost energy!

Obviously not a desired effect, so the solution was to only draw the missiles into planes 2 and 4, and collision detect only to plane 6. This gave quite a lenient feel to the foreground collision detection as colours 0,1,2 & 3 in the foreground graphics would not register a hit. There was many a sprinkling of the other colours in the graphics, though, so the effect was not too noticeable.

As a hint, if you want to see the sort of effect that using only a single plane of some graphics would have in collision detection, load your graphics into *Pixmate* or *Butcher* (image processing programs) and switch off the bitplanes you will not be using. This gives you a visual indication of just what is going to register a hit. It also gives you the opportunity to swap a few of the colours around to get the best use from the single plane.

Being Generous

It is usually preferable to make collision detection 'generous'. This simply means that the player can stray into objects by a few pixels before a collision is detected. This is not possible to do if you are using the hardware collision detection, but is quite simple if you are implementing it in software.

Figure 1 shows the main sprite from Level One of *Blood Money*. Beside it is its collision mask. Rather than use the actual mask of the sprite, a new mask was stored that had a few pixels trimmed off around the edges. The blitter, rather than the hardware method, was used to AND this mask with a plane of the screen.

This allowed certain parts of your helicopter (note in the diagram the blades have been removed) to enter the side walls with safety. This allows a little more skill into the game and eggs players on to have a go at squeezing through tight spots.

This method of collision detection has one major downfall. Although we may detect we have hit an alien, we do not know which one. For *Menace* this does not matter as the game was kept nice and simple, with collisions with any alien simply reducing your energy – the only exception being the bonus icon that you shoot and pick up. The bonus icon, though, can only appear on its own on the screen, so if we hit an alien during the bonus routine, it must of course have been the bonus icon.

What hit What?

The last part of the collision detection, missiles to aliens, must be able to determine what particular missile hit what particular alien. This must be implemented using some

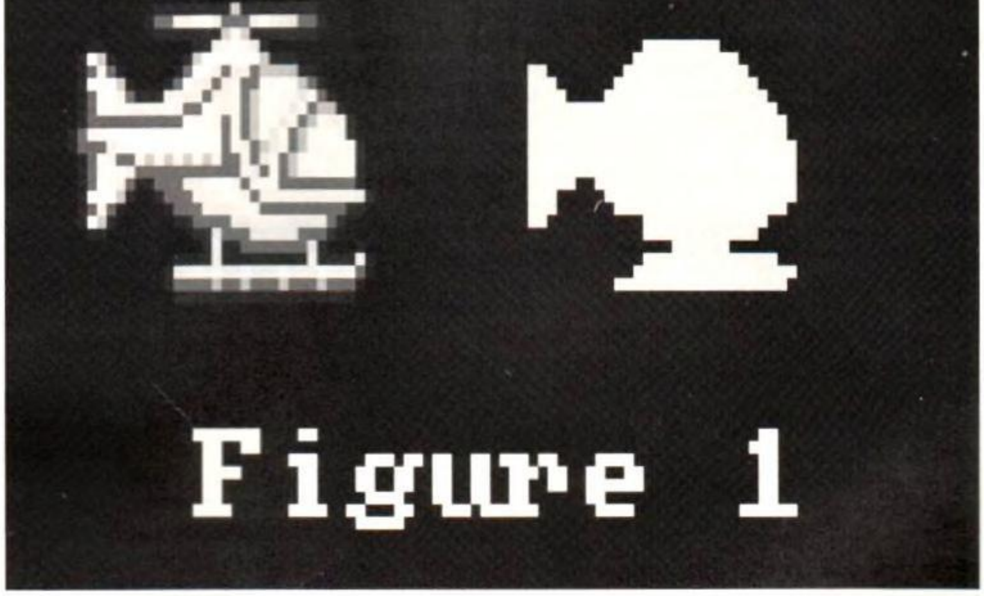


Figure 1: The main sprite from *Blood Money* with its mask for collision detection. The mask is smaller and has no rotor blades, to encourage the player to squeeze through small gaps.

form of coordinate checking. This is the second main way to perform collision detection.

Coordinate checking can never be pixel perfect unless we checked every pixel coordinate in the missile to every pixel in an alien, which would require HUGE amounts of time and cause a game to crawl along. What normally happens is that a 'box' is defined within the two objects you are checking. A box only needs two coordinates, the top left and bottom right. If two boxes overlap then we flag a collision. You can see this is quite rough in that not many aliens appear as a box shape.

In *Menace* the missiles are quite small, so rather than define a box for each missile, only one coordinate at the tip of each missile was checked. If this coordinate entered the box of any alien then a hit was registered.

I made all the alien collision boxes 32 x 24 pixels in size, exactly the same as the alien size. This would appear to cause problems if the alien was quite small in size, not filling the entire box. Have a look at Figure 2. This shows a small alien with a missile entering its box, flagging a collision. But as you can see we would not want this particular case to flag any collision as the missile is quite far away for the actual alien itself.

One way around this would be to store a collision box size for each alien. The one in Figure 2 for example would suit a box of 24 x 18 better. This makes the calculation of whether a missile was in a box or not more lengthy, though, as we are not dealing with constant numbers. This may seem being picky but if we have 12 aliens flying around, and as you can have up to 18 different missiles active at once (missiles, laser, canons etc) this would mean checking every missile against every alien = 216 checks. This will be quite time consuming as there is a fair bit of calculation in each check.

Make it Simpler

What we need is a quick way to tell if a missile has hit ANY alien; if this happens we can then check for WHICH alien it has hit. Fear not, for this is quite simple to do.

What we need to do is combine the mask detection with the coordinate detection. We cannot use the Amiga's hardware detection for missiles to aliens as this only works with hardware sprites. The blitter can do a quick check for us, though.

The blitter has a bit called BZERO in the DMACONR register. This bit will be set true if the result of the previous blit operation is all zeroes. We therefore get the blitter to perform an AND operation between the mask of the missile, and plane 5, which only contains data from the aliens. If the BZERO bit comes back false, ie the result of the blit was NOT zero, then the missile must have collided with some alien data.

NOW we can perform the coordinate detection to find out which alien we hit. This also solves the problem in Figure 2, which would not now flag a collision because the mask detection would not flag a hit, so therefore the coordinate detection would not be then be performed.

When we perform the blit to AND the mask with plane 5 we switch off the blitter Destination channel. We do not want the blitter to draw the result of the blit, so it is pointless to have an area of memory to point the D channel to.

It is quicker and therefore more efficient to switch off the Destination channel for this blit. This is a handy feature of the blitter allowing us to combine a few channels in some way, but only being interested in whether the result was zero or not.

This method of performing the missile-to-alien collision will be considerably faster, even taking into account the blitter operation. On average only three out of the eighteen missiles would actually be hitting aliens. This reduces the coordinate checking to a lower level: only 3 x 12 = 36 comparisons.

Love Missile F-111

That, then, is all the collision required in *Menace:* ship to aliens, ship to foregrounds, and missiles to aliens. The source code for this month (on next month's disk) covers all this, along with the drawing of the missiles. There are four types of missile:

- Normal missile
- Cannon
- Laser
- Outrider

All have different ranges and speeds. They all simply subtract one from an alien's hit number, but lasers are not killed once they have hit an alien – they carry on till the end of the screen – and are therefore more powerful. All other weapons are killed as soon as they hit an alien.

Graphically they are quite small (typically 16 x 4 pixels) and are just four-colour. The blitter is used to draw them into plane 2 and 4. Weapons in shoot-em-ups nowadays are getting BIG. If you plan on trying to write one, I would say the bigger the weapons the better for MEGA effects. There should be no difference in the collision detection, only in the extra time taken to draw bigger, badder weapons.

Completing the Code...!

Sadly, there was no space for the code on this month's Coverdisk, but watch out for it next month – the *Menace* code is now pretty well complete and playable!

All the paths for Level One are there to be blasted, collisions to aliens will reduce your ship energy until zero, at which point you will be returned back to the CLI. When you reach the end of the level the game will also return to the CLI.

This leaves only the guardian to be added, which will be covered next month along with a discussion of the ancillary stuff such as music (where does it come from, how much memory should you reserve, how much does it cost! etc), text routines, disk routines and such like. ■

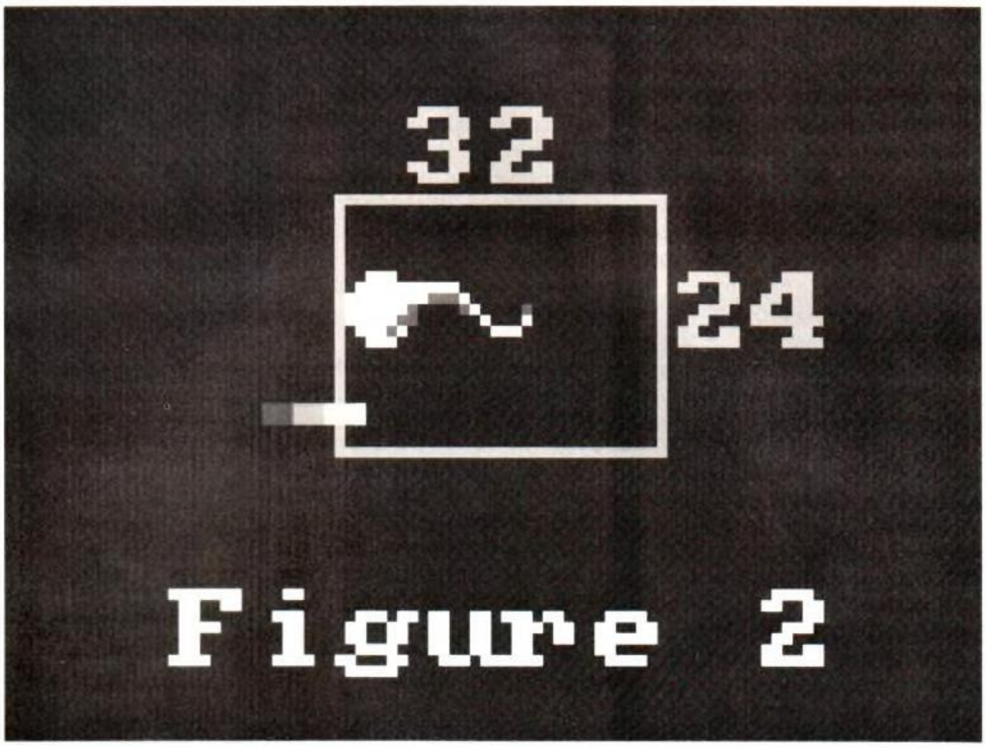
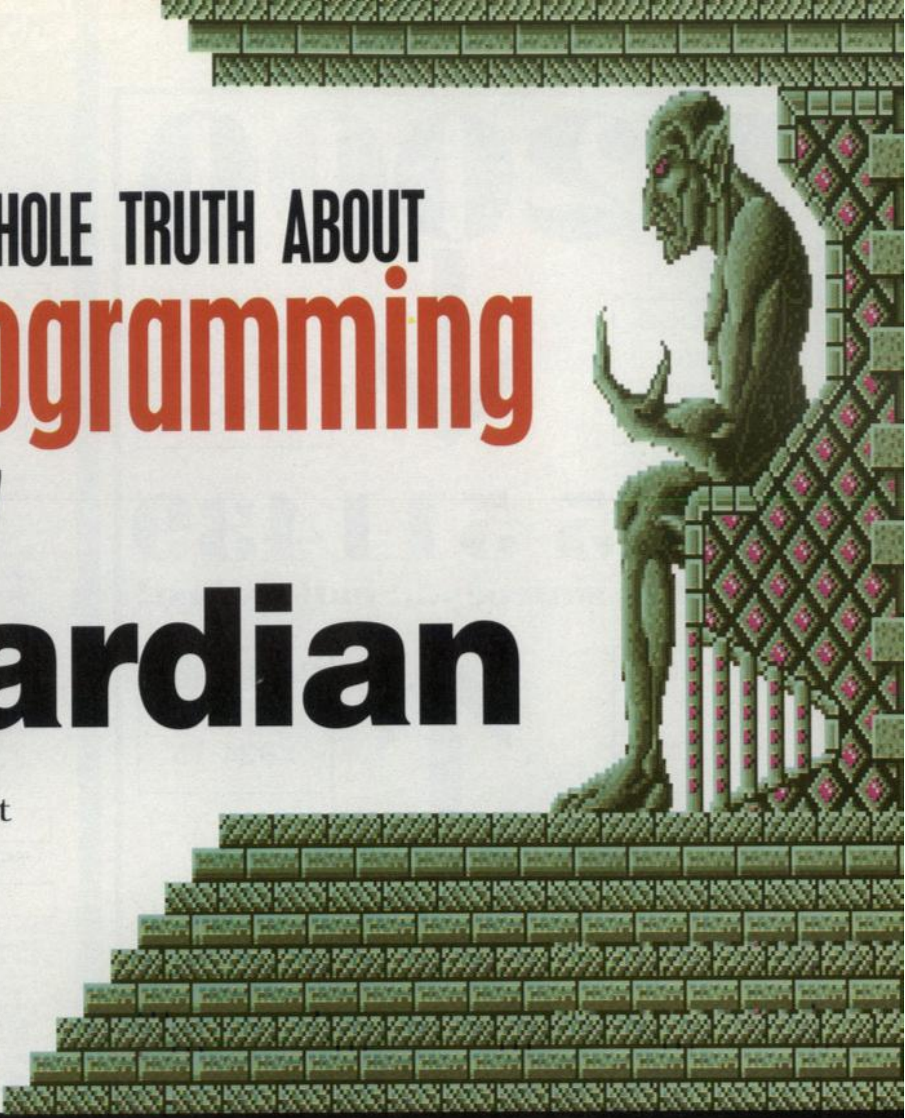


Figure 2: An alien from *Menace* is struck by a missile. If a standard-sized box is used to define the aliens' collision area, this rather small alien will be hit too easily. A smaller size of box is needed for smaller aliens.



THE WHOLE TRUTH ABOUT games programming PART 7

The Guardian

Here it is – the last instalment of *Menace, the game*! This month's source contains the addition of the guardian graphics and code.

The guardian is simply made up of a few normal aliens, as described last month. It is not normally feasible to have a huge animated end guardian as it would require vast amounts of memory. The usual sacrifice is to have the main bulk of the guardian as a single bitmap, with bobs or sprites overlayed on top for the animating sections.

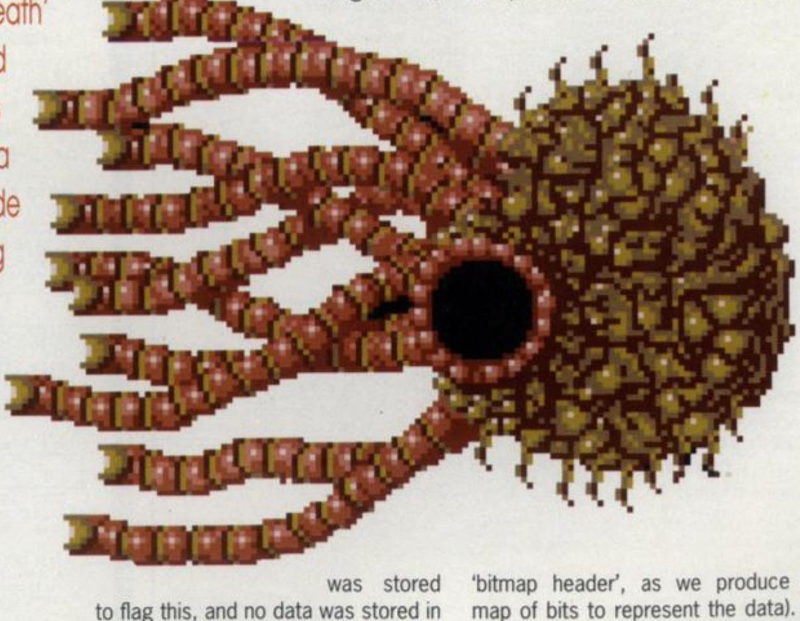
The classic *R-Type* did this in the end of Level One guardian where only the tail and a small part of the stomach were actually animated, but it was still pretty impressive. *Menace* is not that impressive, but it does demonstrate the usual technique.

Big Bad Boys

The pictures above and right show the guardians from Levels One and Five respectively. These are the *DPaint* screens. All guardians are 256 x 192 pixels in size. They are drawn on the right-hand edge of the screen in strips of 16 pixels, just as the map was. Rather than store them as a simple bitmap image (3 planes x 192 high x 32 bytes wide = 18432 bytes) a simple compression was used in order to save memory.

Each strip of 16 pixels was compressed by noting where all the zero words ocurred and only storing the actual non-zero data. This was done by looking at the words from each plane of the image in succession from the top of a strip down to the bottom. If all three planes held no data (this happens a lot, as you can see from the figures) a single bit was stored to flag this, and no data was stored in the compressed file.

“Then the 'death' path is initiated which ensures nobody takes a leisurely attitude when dealing with the guardian!”



If any of the planes did contain data then this was flagged by a single bit, and the three words of data were copied to the compressed file. At the end of the strip (192 lines high) we will have only copied the non-zero data to the compressed file, and will have 192 bits of data (24 bytes) to signify which line of the bitmap this particular data came from. This is a very simple but relatively quick compression method (known as a 'bitmap header', as we produce a map of bits to represent the data). It usually halves the size of the guardian data for each level.

Eye Holes

You can see in the picture above the 'hole' for the eye in the Level One guardian. The eye is simply a normal alien following a standard path. The game knows when a level has been completed when this alien is destroyed. All aliens have a unique number so this is very easy to check.

The guardian bitmap image is drawn in the front playfield so all the aliens appear behind the image as with the foreground scenery. This allows aliens to 'appear' anywhere on the screen, but as long as they are behind the guardian it will not be noticeable. This is how the small 'tadpoles' on the Level One guardian are repeated. Their path data simply makes them appear under one of the guardian tentacles, then swim left till they are offscreen, then go back to the tentacle; and so on. Nice and simple, but it works.

The guardian path is repeated for about 30 seconds, which should be enough time to kill the guardian. If it has not been killed in this time then the 'death' path is initiated in which case all the aliens are substituted for homing mines that cannot be destroyed. This ensures nobody takes a leisurely attitude when dealing with the guardian!

Death by Explosion

When you finally kill the guardian another alien path is started. This one, though, is not deadly, but is simply a collection of explosions all around the guardian body to give the effect of it exploding. This is no different from any other path data and shows that a flexible routine can be used in many places in a game, saving the task of writing more code for some effects.

And that is basically all the game ingredients covered. I have not presented the code for the 'extras' that go into a game as many are quite simple and others have been are the result of many months' work and cannot be published, but I'll run over some of the main ones.

Disk Routines

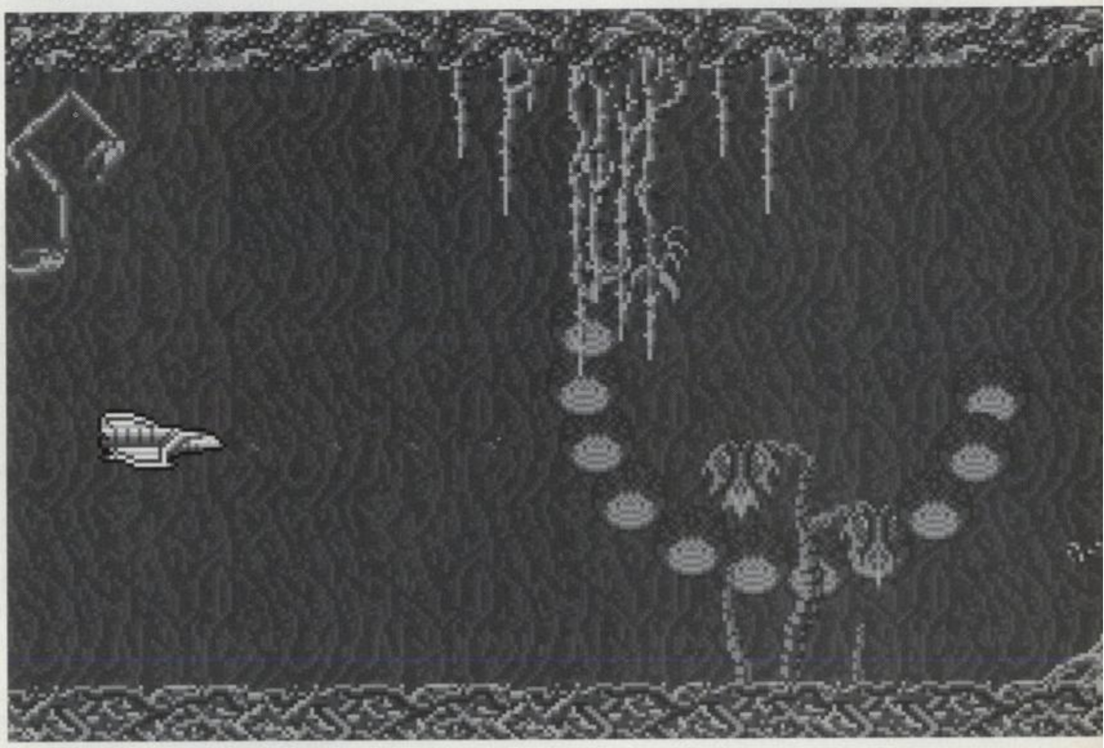
Hmmm, the main reason for many sleepless nights for Amiga programmers! There are three basic levels to using the disk drives on the Amiga for reading/writing your data. AmigaDOS is by far the simplest, and is frequently used for development tools.

To use the DOS routines for a game requires that the operating system is fully intact. This causes severe performance and memory loss. The performance loss can be solved by using the framework given in the first article to disable then re-enable the system when you wish to use a DOS routine.

The memory loss due to the operating system, though, cannot be solved. You will typically lose 100 KBytes if you want to use DOS. This is a lot of memory to a programmer so the DOS route is not usually taken.

The trackdisk device is a set of Amiga system routines that allow you to access the disk as individual sectors. It is quite fast and can run with the minimum of the operating system being intact. Memory loss is still a problem, though, at around 50 KBytes, and once again you have to enable the system to use the trackdisk routines. Providng you can work with this it is a useable alternative to the real hardware nitty gritty.

Getting right down to hardware register level is the lowest we can go. Come down to this level and you have complete control of the system and ALL the memory. Be warned though, MFM encoding, Precomp, SYNC words etc. are all tricky issues. In their individual ways they are quite straightforward, but the difficulty is in testing them all together.



You cannot use a monitor such as Monam2 to debug, as this would require the system to be running, which would interfere with the code you are trying to test. The ideal situation is remote debugging (connecting two Amigas via the parallel port, as Devpac Professional allows) but this is quite expensive.

The method I used when first writing the disk routines was simple trial and error and many late nights. Luckily they only have to be written once. Once they are working simple refinements are all that is required. The Abacus book 'Amiga Disk Drives Inside and Out' has recently appeared on the market, which should prove to be a big help.

This is the only method to use if you want the full memory and complete control so it is well worth spending time writing some *reliable* disk routines for your game.

❝ a flexible routine can be used in many places in a game, saving the task of writing more code for some effects❞

Music And Sound Effects

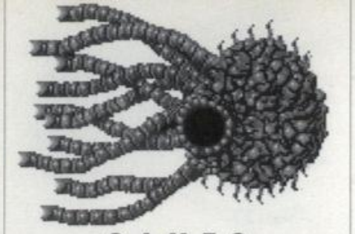
The music and sound effects for *Menace* were written by Dave Whittaker. His name is fairly well known for Amiga game music (other titles include *Shadow Of The Beast* and *Xenon II*). The ideal situation is to get the music written by someone like Dave, who does this for a living.

At the end of the day what you get for your money is the music and sound effects, for basically any machine you require, along with the code to play them, all supplied in a single module of data.

You simply call one of his routines from within your code, and off goes the music or sound effects etc. This makes life very simple for the game programmer, the music and effects usually only taking one day to be added to the game.

The Amiga, though, is well catered for in the music department. You may decide to write the music yourself with one of the standard packages such as *SoundTracker* or *SoundFX*. These popular programs have source code available to play the music that you create. You can therefore save some money by having a friend with a little musical ability, compose a track on one of these packages, then spend a little time on getting the player working in your own code.

These packages do not, however, cater for the playing of sound effects over the top of the music. A simple



sample player is all that is really required to generate some effects, so the simple solution is to allow the game to have music or sound effects going, but maybe not both at once.

The main minus point about using a standard Amiga package is simply one of portability of the music. If, for instance, an ST version of the game is required then ideally the music writer and player you use on the Amiga should also be available on the ST. I believe that there is now a *SoundTracker* file player for the ST, so if you use that program this would solve your portability problem.

Memory wise, you should leave aside about 100 Kbytes for a decent soundtrack. It is possible to have a soundtrack that does not use samples, but generated instruments as used on the C64. Soundtracks that use this technique will quite easily fit within 10 Kbytes, but you obviously lose a little of the effect of good Amiga samples.

One point to bear in mind on the music front is one of playback speed. Most of the player routines are patched into the vertical blank routine, so on PAL machines the music is updated 50 times per second. This seems fine, but remember that the NTSC standard, as used in the States, updates 60 times per second. This means that the music will be 20% faster over there than over here. This can turn a good soundtrack into one that sounds a bit naff because it is too fast.

Your game should either use a timer to generate a 1/50th of a second interrupt, which will be the same on any machine, or detect which type of machine you are on (PAL/NTSC) and slow down the music accordingly. This is simply done by not calling the music routine every 6th vertical blank on an NTSC machine. Remember also that you have 20% LESS processor time per frame on an NTSC machine. This can cause havoc if you write a game, designed to run in a single frame, that has not taken into account this loss of processor time.

All new Amigas with the fatter AGNUS chips now have the capability to be switched into 60 Hz mode. Monitors (and some TVs) can handle this, and ideally you should try to get access to one to test out your code on in 60 Hz mode. All Amiga games should also now start to accomodate this 60 Hz switch capability on a key press in a game.

Running your game in 60 Hz mode will result in the game playing 20% faster, and also filling the entire PAL display (even though the graphics may look a little stretched). ST owners have always had this capabilty, and it is a nice feature to include into a game.

Intro Sequence

A nice rolling intro for the game can do wonders for its appeal. They add nothing to the game, but add a little variety to the package. A good intro to a game should be a piece of code that is technically and visually excellent, the sort of thing that is not really possible to implement in a game itself, but shows of some of the capabilities of the Amiga.

This sort of effect should also be added to the game should anyone eventually complete the game. It is a real letdown when you spend months playing a game, finally manage to finish it, and up pops a bit of text saying 'Well Done'! Programmers who do this should be shot (there is an animated sequence at the end of *Menace* and one after *Blood Money*, by the way, so I'm safe from the assassins for now!)

"Games like *Populous* and *Stunt Car Racer* really come into their own in head-to-head mode – they have cost us many a day's work!"

AND FINALLY...

TEXT ROUTINE

Try to design an impressive character set, maybe incorporating some copper tricks to jazz the text screens up a bit. If you are working on a monitor then keep in mind the TV users and don't have a small character set, they will probably not be able read it.

HIGH SCORE TABLE

Most arcade style games need a high score table. They are usually pretty boring to write (this is usually the first routine to get delegated to new programmers!) A 'save to disk' option is usually a must.

DEMO MODE

Any game that has a self-play demo mode is much more likely to be loaded and displayed in a shop. It will also help when the game is shown at the multitude of computer shows throughout the year.

SERIAL/PARALLEL LINK

Many games are now incorporating these types of link for head-to-head action. Games like *Populous* and *Stunt Car Racer* really come into their own in this mode – they have cost us many a day's work! It really depends on if the game is suited to a two-player head-to-head, of course, shoot-em-ups generally are not, but there is always a first time for everything...

PROTECTION

Most programmers tend to implement various protection schemes of their own, although the disk duplicators can often offer their own techniques also. In my belief it is not possible to protect a disk 100% from being copied. The main aim is to DELAY as much as possible the inevitable 'cracked' copy appearing.

Most games tend to sell their strongest in the first month. Over a period of two years, 80% of the sales may well have happened in this first month. If you can therefore stall the pirates for as long as possible, people have a much better chance of seeing the game in their local shop, than suddenly appearing in the post from friends.

Adding protection can be a long and tiresome process. It is sad that it has to be done, but that is a topic we are all familiar with...

On that sombre note I'll wrap up this series. I hope many of you have a go at some programming. It is possible to write games in your spare time as a hobby. You never know, it could lead to a full-time job. And as the old saying goes, "A man whose hobby is his job, is a very happy man" (circa. 1990 Dave Jones, DMA Design).