BRITAIN'S LEADING MAGAZINE FOR AMIGA OWNERS

AMIGA FORMAT

ISSUE 7 / FEB 1990 / £2.95

GRAPHICS
Tricks of the trade from the master artist

PROGRAMMING
Menace and Blood Money author tells all

MUSIC
You too can become a keyboard maestro

SECRETS REVEALED

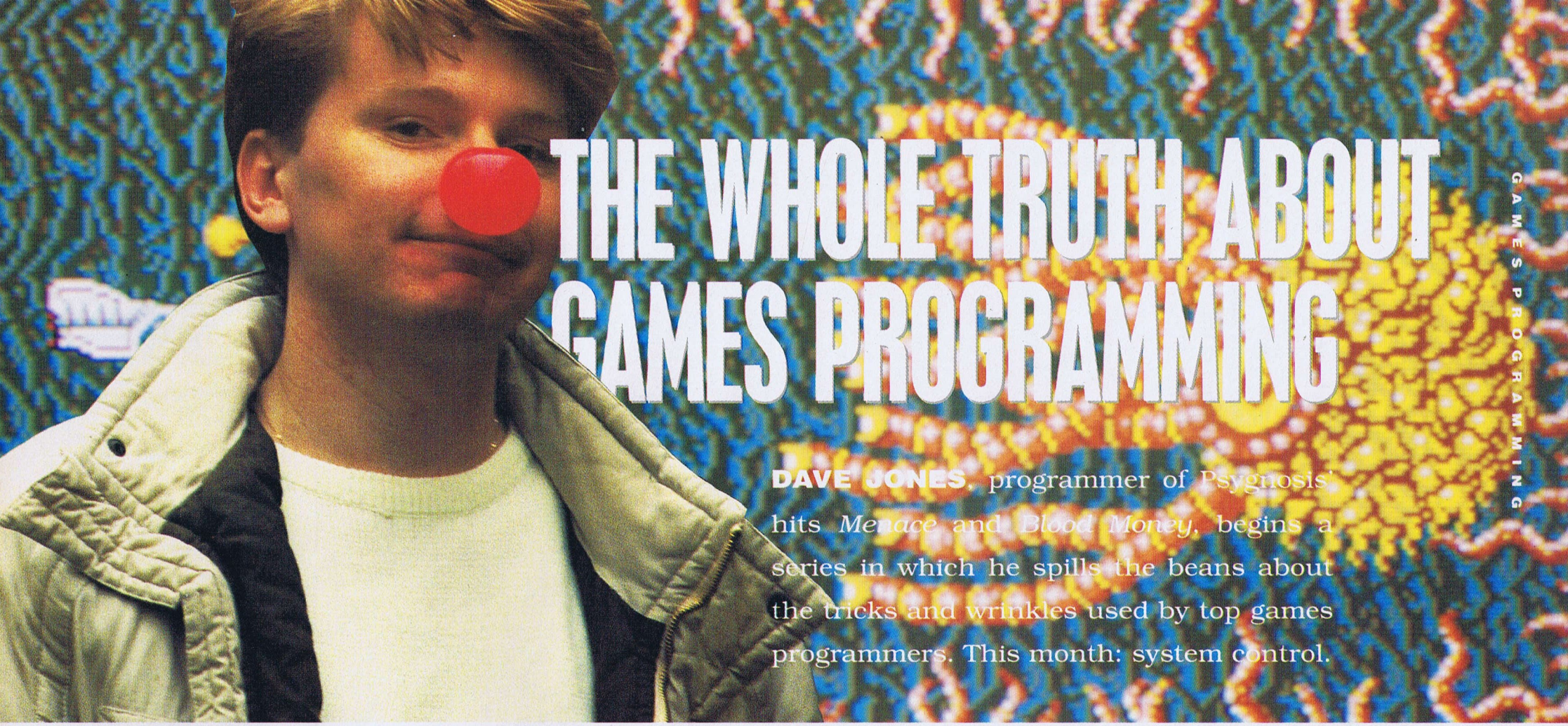
The experts help you to get the most from your Amiga

NO AMIGA COVERDISK?
DEMAND ONE FROM YOUR NEWSAGENT NOW!

BEGINNERS
Indispensable guide for the new Amiga owner

GAMES
Definitive reviews of all the big games

9 770957 486004 02



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...

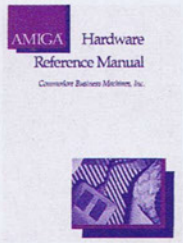
Dave Jones is now 23 years old and lives in Dundee, Scotland. His first game, *Menace*, was released by Psygnosis in November 1988 to considerable acclaim from reviewers. It may look somewhat dated now, but many of the programming techniques it uses are extremely advanced.

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...

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.

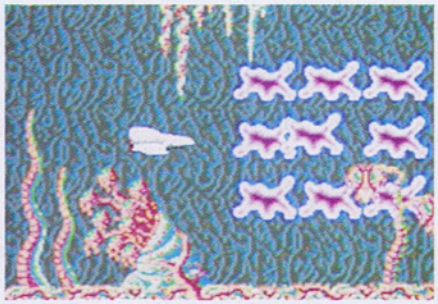
MASKING – When drawing graphics into the screen it is preferable to leave intact the graphics that are already there. This is done by masking, which lets all 'holes' in the graphic that we are drawing show the graphics underneath. The blitter in the Amiga is an expert at doing this for us.

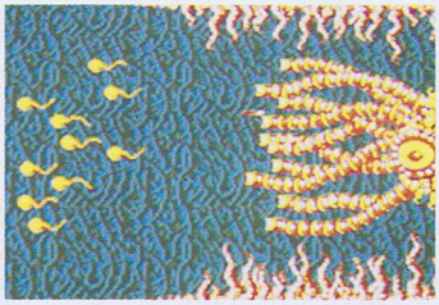
EDITOR – Not a text editor, but a piece of software that allows the editing of game data such as level maps, or alien movement patterns. These are quite time-consuming to write but save a lot of time once completed. Menace has no editors: it was the first game I had written, and all data was typed in 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 (VD0:) 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.

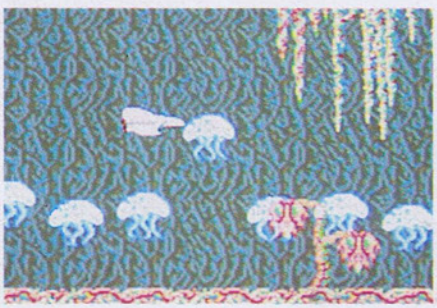
Note that we did not clear the screen in Framework, yet it did disappear. This is because we turned all DMA (Direct Memory Access) off. The Amiga uses DMA extensively when it requires to fetch or move memory. All the custom chips use this feature to fetch the data they need (blitter, sound, sprites etc)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

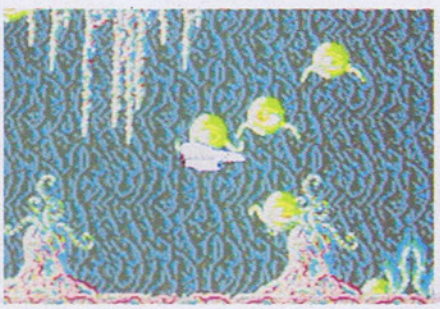
THAT MENACE SOURCE CODE...

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

TwoBlanksbsr
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:

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.

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. ■