BRITAIN'S BEST-SELLING MAGAZINE FOR AMIGA OWNERS

AMIGA FORMAT

ISSUE 11 ■ JUNE 1990 ■ £2.95

DISCOVER NEW WORLDS

Explore the power and beauty of fractals, where chaos rules

COVER 11 DISK

TOWER OF BABEL

PLAYABLE DEMO OF RAINBIRD'S IMMENSE FORMAT GOLD GAME

NO AMIGA COVERDISK? DEMAND ONE FROM YOUR NEWSAGENT NOW!

GAMES YOU SIMPLY *MUST NOT* MISS

■ F29 Retaliator ■ Gravity ■ Their Finest Hour ■ Tower of Babel

FOUR FABULOUS FORMAT GOLDS

OVER 20 games reviewed ■ ESSENTIAL guide to spreadsheets ■ [illegible] PRINTER do you need? ■ Review of the NEW Amiga 1500 ■ The books you NEED to read ■ How to BEAT *Rainbow Islands*

06
9 770957 486004

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

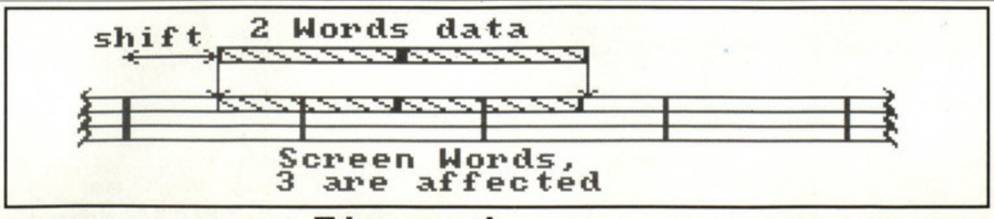

Figure 1

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.

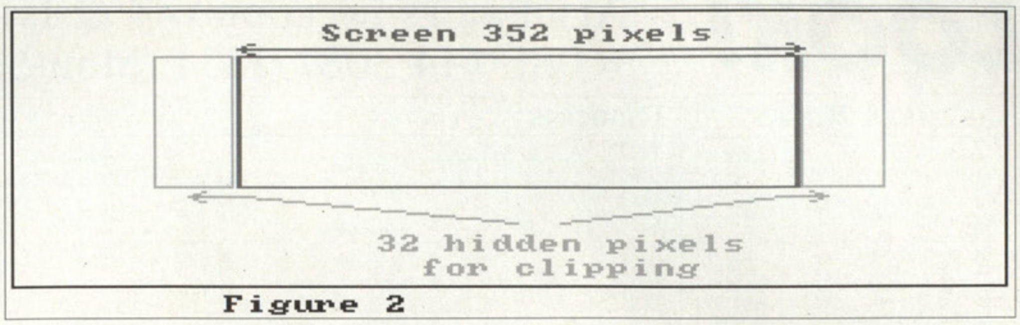

Figure 2

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!