Your Turn! Part 2

Bullfrog's coding methods revolve around making life as easy as possible, so giving more time to pose in rather ridiculous positions for press photographers. Or not, as the case may be.

How to program your own games in assembler

Part 2 of a continuing series which aims to teach you how to program like the pros. Bullfrog coder **Scott Johnston** delves into the mysterious world of player control, and how to tweak a sprite'.0's movement to give an impression of inertia and momentum. So if you want to avoid stilted, lifeless characters, read on...

Welcome back. If you followed last month's article, well done. If not go back and do so. This month we're going to tie the movement of our sprite to the joystick. If you run Demo you will find the man now runs left and right with momentum. The exit key has changed to Q because when Escape was pressed you got the repeat of your last input. Does the program work? Great.

Now let's make him run up and down the screen as well. If you look inside Move.s you will notice two new things. At the top of the file is a small list of equates. These control the graphics for our man and the speed he accelerates and moves. Try changing the two equates for the speeds. The man should move differently depending on the numbers you put in. If you change the animation equates the man could start doing strange things. Another example of equates is in Draw.s on the two asr.w lines, with the word FOUR. This is easier to understand than placing a 2 on the line.

Continued overleaf

APOLOGIES

Our sincere apologies for the lack of an IFF to binary convertor program on this month's Coverdisk. This was entirely due to lack of space on the disk, and we apologise to all the coders straining at the leash to put their own graphics into the demo. Never mind, you've got another month to make them look even better than they are now!

Anyway back to Move.s. After this we have the subroutine _move_all. Last month we placed all our code here, but now we will place our code into new subroutines which _move_allcalls. Our man now has 5 variables. These are as follows:

`man_x` (whereabouts he is horizontally on the screen.)

`man_y` (whereabouts he is vertically on the screen.)

`man_vx` (the speed and direction in which he is travelling horizontally.)

`man_vy` (the speed and direction in which he is travelling vertically.)

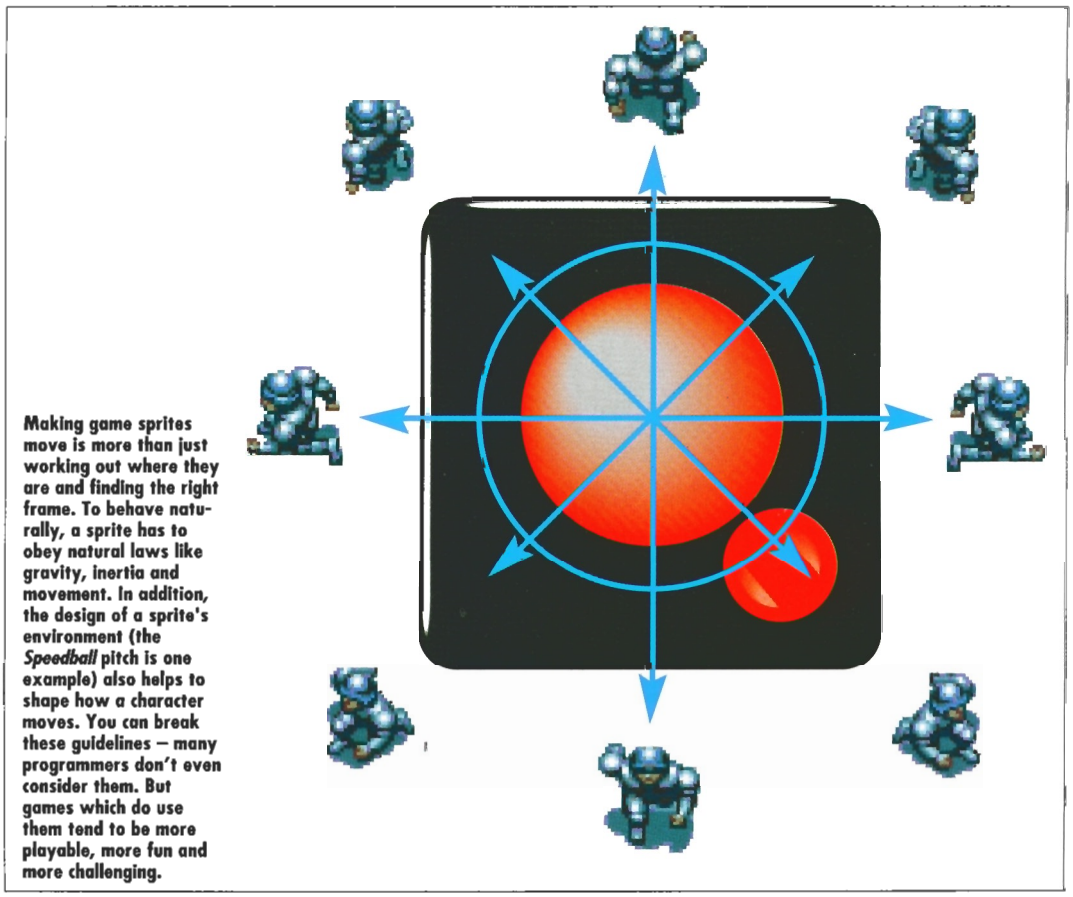
`man_frame` (The current frame of animation that is displayed.)

Inside the first of our new routines _man_x is the code to read the joystick, and move the man left and right. Study this. Now if you copy this and place it into _man_y, change the variable names to their Y counterparts and then change the following lines:

`cmp.w #300*4,d1` to `cmp.w #176*4,d1`

`move.w #300*4,d1` to `move.w #176*4,d1`

and place an asr.w #FOUR,d1 into draw.s at line 14. Now you should have momentum on the Y axis. Try it. Did it work? If not go back and study the code again. You probably made a small mistake somewhere along the line.



Making game sprites move is more than just working out where they are and finding the right frame. To behave naturally, a sprite has to obey natural laws like gravity, inertia and movement. In addition, the design of a sprite's environment (the *Speedball* pitch is one example) also helps to shape how a character moves. You can break these guidelines – many programmers don't even consider them. But games which do use them tend to be more playable, more fun and more challenging.

Right, now change the names of the dot labels (those that start with '.') to what they are actually doing. Good, now did you under-stand that code? If so see if you can change it so that instead of the man stopping when he reaches the edge of the screen, make him bounce off. Basically you change four commands from a Move.w to a Neg.w. If you did not understand it, don't worry because I will explain now precisely what it is doing.

To start with we pick up the velocity of the man, and the direction in which the joystick is being moved. The joystick will be a 1 if the joystick is going right, 0 if it not moving, and -1 if it is going left. We then multiply the joystick direction by the acceleration and add it to our current velocity.

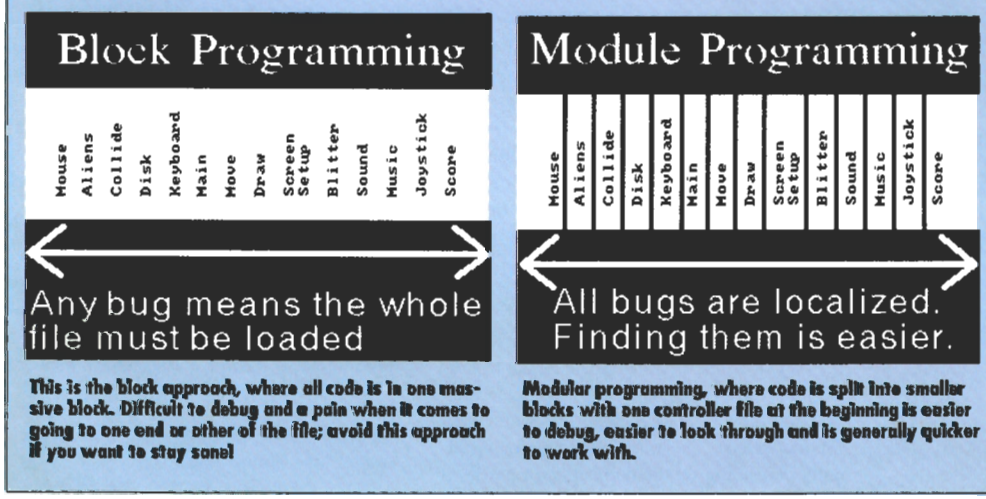
Now the velocity needs to be checked to make sure that we are not going faster than we want in either direction. First start off by checking in the right-hand direction, and if it is going too fast then scale it down to the minimum speed. Then check the other side and scale it down if need be. OK that's the velocity just about done. We now get our current x position and add the velocity to it. Check the new x position with the left-hand side of the screen, and if we have gone off then move us back on to the screen and stop the velocity.

Test the other side of the screen, and again stop the velocity and move it back on if it has gone off screen. The final piece of code we have in this routine is the slow down. This checks to see if we are moving the joystick, if we are then we don't want to slow the man down. After this try and find out in which direction you are moving. If you're going left we want to increase our velocity to slow it down. Sounds strange but the maximum left speed is in fact –16, or whatever you change MAX_SPEED to. If you are going right you want to decrease our speed. The reason for subtracting 2 and then adding a 1 to the velocity is so that you don't have to branch past the going-left section. Follow that OK?

Now wade through the code in _man_anim and see if you can work out what is going on.

Continued on Page 142

That's the way you do it!



This is the block approach, where all code is in one massive block. Difficult to debug and a pain when it comes to going to one end or other of the file; avoid this approach if you want to stay sane!

Modular programming, where code is split into smaller blocks with one controller file at the beginning is easier to debug, easier to look through and is generally quicker to work with.

Some people last issue were confused as to how you go about using the Bullfrog development system. Well the idea is that the code is controlled by one file, which loads in all the others. This control file is called Demo.s. So as you make changes to other files, you must save them. Then load and assemble Demo.s to see the results.

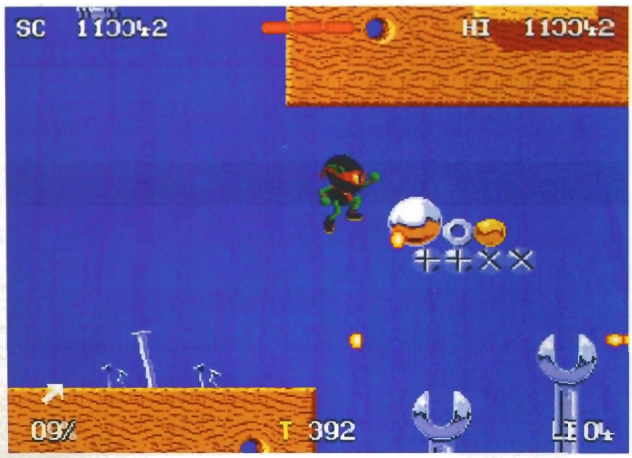
This system has a lot of advantages. Programmers build up a collection of small programs and odd utilities which they use time and again. There's no point keeping them all in the same file though. You might only use one of these routines, and all the others are stuck to it, wasting memory and disk space.

Hence the need to split programs up into many small files. Once assembled, all the code will be in one block anyway. This approach to splitting up programs into small chunks is known as modular programming. Although assembler is not well suited to this approach in some ways, it still makes sense to keep routines in separate files.

NEW COMMANDS

```
Neg.w    d0.
```

This reverses all the bits in d0. This is a great help when you want to change the signs of numbers. For example 5 becomes –5, and –9 will become 9.



***Zool* is one of the best examples of movement control tweaked to perfection. The main character's behaviour has been developed extremely well.**

Myth is one of many games where the control leaves a lot to be desired. The main character moves, but too much time is spent moving. More speed!

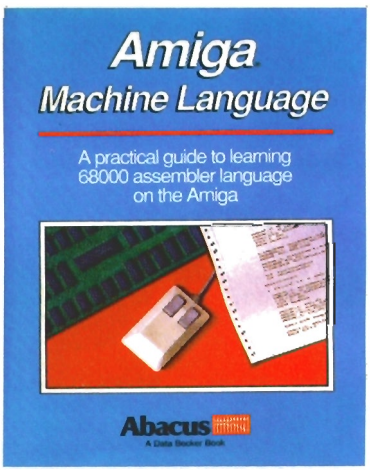
You will have noticed that there are loads of other files which we never really touch. The only ones we have used so far are Move.s Demo.s and Draw.s. If you have looked at Demo.s then you will see that loads of things are included from here. Most of these files are for various set ups, ie read the joystick. Others are called from various places inside our code, for example the sprite draw section. One of the more important of these files is Display.s. If you have a look inside Display then you will see the main loop of our program. It starts off by waiting for the vbi. The vbi, or vertical blankinterrupt happens every 50th of a second, and is the time in which the videobeam is travelling from the bottom of the screen, back up to the top.

We then clear the screen, this is so that we don't have the previous screen display at the same time. Clearing the screen is not always the quickest way of doing things, for example you could remove the sprites, and then redraw them. Draw All calls the routines inside of Draw.s, and this is where we draw everything, the man, the blocks, the objects, everything. We then swap the screens. The system you are using is called double buffering. This is where you see one thing on the screen, while you are drawing the next scene on another screen. The screens then swap places, and we draw the next scene (scene 3) on the first screen. Swap places again and so on.

This prevents the graphics from flickering when they are drawn. We then go on to the movement routines, and here we move everything, advance the animation counters, detect for collision, kill off the hero if you can. Check for a key press, if it hasn't been pressed, then loop around and do it all over again. Otherwise exit from here, turn everything back to normal, and terminate the program. That's just about it for this month. Next month we will start to make our small program into a platform game. This includes the need for gravity. See if you can change the routine for man_y so that gravity plays a part. Good luck.

For the animation of the man you want him to look like he is actually going the way he is facing. There are several ways in to do this. The best way is pick up the current frame of our man. Then have a look at the man's velocity, if this is a zero it means he is not moving, therefore you'll want to display the man facing towards the screen. If the number is negative then the man is running to the left. You have four frames of animation for our man in both directions, so check that the current frame does not exceed the allowed frames, or that if it does you set it back to an allowed frame. And that's it for now... see you next issue.

RECOMMENDED READING

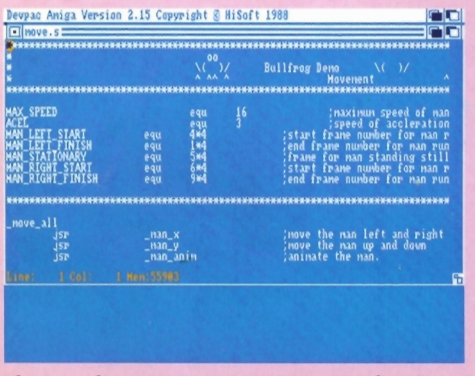


***Mastering Amiga Machine Language* by Stefan Dittrich is one of the easier tutorials to get to grips with. Other good reads: *System Programmers Guide* (also Abacus).*Programming the Z80* by Rodney Zaks and *The Hardware Reference Manual*, published by Addison Wesley.**

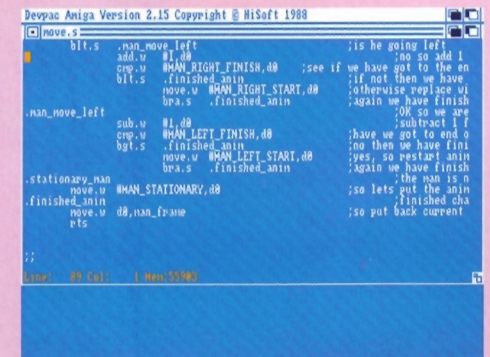
Using this month's code

This month, due to an especially tightly packed Coverdisk, the code has to be dearchived on to a copy (a copy, mark you) of the *Devpac* disk from last month's issue. In order to get at it, you will need two blank disks for the *Legend of Valour* and *Chaos Engine* demos. Don't give me hogwash that you don't want to see them – they are of the quality every programmer should aspire to.

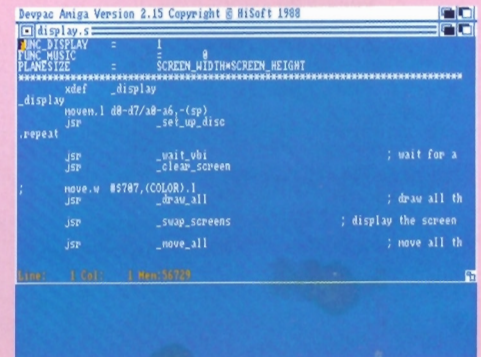
Anyway, just boot up the Coverdisk but be sure to have two blank disks plus a copy of *Devpac* handy. After the two demos have been decrunched (which takes only a few minutes) you will then be prompted to enter the *Devpac* disk. Another minute and all the new versions of code will be dumped on to it, ready for you to use.



These are the equates in Move.s that can be altered to give different movement characteristics to the main sprite. Feel free to experiment.



Here are the dot levels. Study the notes on the right very closely before you alter anything of the code. Assembling the changes should help understanding.



This is the main loop of the program, held in the file display.s. Be cautious about changing this area of code, as the results can be very far reaching.