

The complete guide to games programming with AMOS

AN **AMIGA**  
FORMAT BOOK

# Ultimate AMOS

Jason Holborn



Includes a 3.5-inch disk  
with all the code in the book  
plus working AMOS games



Covers AMOS Pro  
Compiler

Easy AMOS • AMOS 1.35 • AMOS Pro • AMOS Compiler/Pro Compiler • AMOS TOME • AMAL • CText • D-Sam

# Ultimate **AMOS**

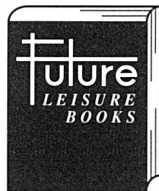
The complete guide to games programming with AMOS



# Ultimate **AMOS**

The complete guide to games programming with AMOS

Jason Holborn



## **Ultimate AMOS**

Copyright © 1993 Future Publishing

All rights reserved. No part of this book may be duplicated, stored in a retrieval system or used as part of any other book, database, program or other commercial application without the publisher's express written permission.

**Book design and production** Rod Lawton

**Cover design** Rod Lawton

**Cover illustration** Paul Kidby

**Author** Jason Holborn

**First published in 1993** by Future Publishing, Beauford Court, 30 Monmouth Street, Bath, Avon BA1 2BW

**ISBN** 1 898275 02 5

**Printed in the UK** by Beshara press

### **Acknowledgement of copyright and trademarks**

This book contains copyright or trademark product names owned by the companies which produce them. Description of these products without mention of their legal status does not constitute a challenge to this status. The author and Future Publishing fully acknowledge such copyright names or trademarks.

### **The included disk**

The programs on this disk have been written for their instructional value. They are not guaranteed for use in any specific situation. The publishers accept no liability for problems arising from their use. The contents of the disk are copyright, and are not to be copied or distributed in the public domain.

---

# Contents

---

<b>Preface</b>	<b>xi</b>
----------------	-----------

---

<b>Using this book</b>	<b>xii</b>
------------------------	------------

---

<b>Chapter 1: Introduction</b>	<b>1</b>
• What is AMOS, what can it do?	
• The AMOS 'family'	
• AMOS utilities	
• Getting started with AMOS	

---

<b>Chapter 2: How AMOS works</b>	<b>21</b>
• The AMOS Editors (Easy AMOS, AMOS 1.35, AMOS Pro)	
• The AMOS Monitor (Easy AMOS and AMOS Pro only)	

---

<b>Chapter 3: Programming principles</b>	<b>39</b>
• Program planning	
• 'Pseudo code'	
• Sub-routines	
• Code comments	
• Code indentation	
• Procedures	
• Handling variables	

---

**Chapter 4: Screens****51**

- Screen modes
- Opening screens
- Screen management
- Screen palettes
- Resizing and positioning screens
- Loading and saving screens

---

**Chapter 5: Screen scrolling****73**

- Screen synchronisation
- Hardware & software scrolling
- Superbitmaps and viewports
- Screen Copy scrolling
- Parallax scrolling
- Continuous scrolling
- Using screen 'blocks'
- AMOS TOME extension

---

**Chapter 6: Screen effects****103**

- The Amiga's co-processor ('copper')
- Copper effects and rainbows
- Screen synchronisation & 'double-buffering'
- Screen compaction

---

**Chapter 7: Sprites and bobs****133**

- Hardware and software 'sprites' (sprites and 'bobs')
- Creating sprites and bobs (objects)
- Displaying and moving objects
- 'Virtual' sprites
- Animating objects
- Flipping objects – 'hot spots'
- Collision detection
- The Object Editor & menus
- The Animation Editor

---

**Chapter 8: Object control** 171

- Keeping track of objects with 'data structures'
- Controlling on-screen objects using a joystick or keyboard
- Setting boundaries for object movement
- Creating a 'bouncing' movement
- More advanced object movement patterns

---

**Chapter 9: AMAL** 223

- Multi-tasking and interrupts
- How AMAL works
- AMAL Editor versus 'embedded' code
- Assigning and handling channels
- AMAL registers — local, global and 'special'
- AMAL instruction set
- AMAL functions
- Using more than 16 channels
- Useful AMAL routines

---

**Chapter 10: Sound and music** 251

- Built-in sound effects
- Sound samplers
- Sample banks
- Handling samples
- The Amiga's sound filter
- Music modules
- VU meters
- D-Sam extension
- Sample Bank Maker

---

**Chapter 11: Games programming** 269

- Games programming principles
- The 'main game loop'
- Game types
- Optimising game code



---

**Chapter 12: Shoot 'em ups** **279**

- Moving the player's ship
- Handling aliens
- Firing missiles
- Explosions

---

**Chapter 13: Maze games** **293**

- Drawing the maze
- Detecting walls
- Picking up objects
- Intelligent 'baddies'
- 'Dungeon Master' clones

---

**Chapter 14: Platform games** **321**

- Drawing platforms
- Tying bad dies to platforms
- Jumping between platforms
- Picking up objects

---

**Chapter 15: Adventure games** **341**

- Designing an adventure
- Writing a parser
- Moving around 'locations'
- Handling objects and monsters
- Adding graphics

---

<b>Appendix A: Useful routines</b>	<b>361</b>
<ul style="list-style-type: none"><li>• Mandelbrot generator</li><li>• ‘Splerge’ effect</li><li>• Parallax starfields</li><li>• Multitasking text input</li><li>• Co-ordinates finder</li><li>• High-score routine</li><li>• Bubble sorting arrays</li></ul>	
<b>Appendix B: Getting your game published</b>	<b>383</b>
<ul style="list-style-type: none"><li>• Approaching a software house</li><li>• Stopping yourself from getting ‘ripped off’</li><li>• Hiding your game’s creator!</li><li>• The PD options – including shareware &amp; licenseware</li></ul>	
<b>Appendix C: Where to go next</b>	<b>393</b>
<ul style="list-style-type: none"><li>• ‘The AMOS Club’</li><li>• ‘Totally AMOS’</li><li>• Magazines</li><li>• Bulletin boards</li></ul>	
<b>Index</b>	<b>399</b>

---

---

## **About the author**

Jason Holborn is a freelance writer and journalist who has worked on the Amiga throughout his journalistic career. His writing career began in 1988 when he joined Future Publishing's then new publication, ST Amiga Format. When ST Amiga Format split into two magazines in 1989, Jason moved across to work as technical editor on Amiga Format, a post he held until he left to pursue his freelance career in 1990.

Jason has been programming a wide range of computers for considerably longer than he has been a journalist, during which time he has worked with just about every programming language – including assembler, C, Cobol and Pascal – although AMOS remains his favourite language.

Jason lives and works in Somerset with a Cockatiel called Cosworth, an XR3i Cabriolet called Derek and an Amiga A1200 called Cyril. His hobbies include fast cars (especially Ford Cosworths), Japanese martial arts, history and culture, listening endlessly to his collection of Depeche Mode compact disks and mucking out his girlfriend's horses!

---

## **Thanks to...**

Special thanks must go to Georgina (my girlfriend) for putting up with me, Nicki, Tammy and Benson, the local Spar shop for keeping me supplied with microwave doner kebabs, my regular drinking partners Barry and Max who constantly remind me what it feels like to have a hang-over, Rod at Future for being so understanding when deadline after deadline passed, Stuart, Marcus and Cliff for the same reason, Richard Vanner at Europress, Dave Smithson for his great ideas, Cosworth for keeping me company, Alfie Noakes for his total lack of jokes, Ross McPharther for his amazing show-stoppers, Derek's mum for being here quite long enough thank you and Depeche Mode for keeping me entertained through those long winter nights.

# Preface

The BASIC programming language is available on every home computer. It's most people's 'first' programming language, combining reasonable programming speed and power with easy-to-understand commands and structures.

The Amiga too has its own standard BASIC. In the early days it was the universally disliked AmigaBASIC. This disappeared from Amiga packs, and was not replaced by Commodore. At the same time, a new type of program appeared. AMOS was BASIC-based, but utilised the Amiga's own custom hardware to provide a programming language which was not only easy to learn and understand, but extremely powerful too.

AMOS has come to be accepted as the Amiga's 'standard' BASIC. Although it can be used to program just about any application under the sun, its main strength (and the reason for its huge popularity) is its abilities as a games creator. The results can be stunning.

AMOS has evolved since its launch. It's got faster, more powerful and more versatile, culminating in the release of AMOS Pro, and the brand new AMOS Pro Compiler. The Compiler converts AMOS programs into machine code, resulting in much faster and more compact code. Nearly as fast, in fact, as code written by the 'experts' in assembly language.

In fact, AMOS now has to be considered the ultimate games creation package. Which is why we've produced what we intend to be the ultimate AMOS guide.

# Using this book

We're assuming you already have some basic (sorry!) knowledge of BASIC programming. Not much, but enough to grasp the principles of variables, loops, conditional operators and other BASIC basics (there we go again). Note, though, that you don't have to be a BASIC expert to use Ultimate AMOS. Far from it.

The first few chapters in Ultimate AMOS are arranged in sequence, starting from the basics and building up to special AMOS features and techniques. After that, though, feel free to dip into the chapters that most interest you when you need the information. You don't have to read this book from cover to cover (although it wouldn't hurt).

To make things easier and more useful, we've included icons in the margins to indicate things of special interest in the text. Here are those icons, together with what they mean:



**Make a note** Most of the things you read get stored away in your head somewhere or other. When you see this icon, though, make sure you store this particular item somewhere prominent. It's quite important.



**Product spotlight** You don't use AMOS on its own. Screens and backgrounds may be drawn in DPaint, in-game music will be written using a dedicated music program, and there are a number of special AMOS 'extensions' which boost its power. Here's where we point them out to you.



**Top Tip** There are lots of ways of saving time, money and effort – and of spectacularly improving results – that you won't find written down anywhere. Except here, that is.



**Clever Coding** Programming is a funny business. For any given end result there are usually half a dozen different ways of programming it. Some are obvious, some are devious. Clever, in other words.



**What does it mean?** Programming is full of jargon. You can't get rid of it, you just have to live with it. But that doesn't mean to say you can't explain it. Which is our rather long-winded way of explaining that we do.



**Warning!** You won't see this icon too often, but when you do, take note! Ignoring it could cost you time, money or your sanity. People often have precious little of any of these, so don't take chances.

## Icons for program listings

We have another selection of icons for the parts of the book where we print program listings. Here's what they mean:



**Type this in** Actually, you don't have to, since all the complete listings in the book are also on the disk supplied. This icon means that the listing is a complete, working AMOS routine. Printing the listings in the book lets you examine the code to see how it works without having to print out a listing yourself from the disk (not everyone has a printer).



**Pseudo code** To program successfully you have to plan what you're going to do. And this is often easier using 'pseudo code'. These listings won't actually work – they're a kind of programming shorthand representing what the program should do and how it should be structured.



**Sample code** This icon means that what you're looking at isn't a complete listing – it is working AMOS code, but only as part of a larger routine. It's intended to demonstrate AMOS commands, functions and techniques.



**Command definition** Speaks for itself really. AMOS commands (and functions) are defined all the way through the book, together with their syntax, parameters (if any) and usage.



# Introduction

- What is AMOS, what can it do?
- The AMOS 'family'
- AMOS utilities
- Getting started with AMOS



**W**hether you're a hardened Amiga fanatic or just a Sunday afternoon games player, there's something intrinsically appealing about writing your own programs. I'm sure that most of us have dreamed of joining the ranks of those rich and famous programmers we see month after month within magazines like Amiga Format and Amiga Shopper. Fuelled by images of bright red Porsches, international recognition and the sort of expensive jewellery than would drown Jimmy Saville, it's not surprising that programming languages such as AMOS have done so well.

Even if you don't achieve the same levels of success as such big names as DMA Design's Dave 'Lemmings' Jones, David 'Elite' Braben or Daniel 'DPaint' Silva, learning to program can be a deeply rewarding experience. Perhaps its got something to do with that feeling of utter satisfaction you get when you finally manage to get that program that you've been working on for the last 2 weeks up and running! What's more, the only limit that programming imposes is your imagination!...

What makes programming languages such as AMOS so special, though, is flexibility. No other software product you could possibly buy for your Amiga can be applied to so many different tasks. Take a paint program, for example. Although you can paint just about any picture that you could possibly think of, you're still essentially producing the same results (a picture).

A word processor is the same – although you could use it to write anything from a love letter to a major work of English literature, you're essentially getting the same results over and over again (formatted text). All fine and dandy, but what else can these applications do? Nowt, that's what!

OK, so this is a little unfair – after all, comparing a programming language to any other kind of software is like comparing a toolbox with a chair – although the tools within the toolbox could theoretically be used to build a chair (providing you have some wood), the chair was designed to fulfil a specific task and no other. (You're probably wondering what all this talk of chairs and toolboxes has to do with programming languages, but the comparison between the two is surprisingly appropriate.)

Like the toolbox, a programming language does nothing more than provide you with a set of tools in the form of general purpose commands – how you apply these tools is up to you. Because the commands offered by the language have been designed to be as general purpose as possible, a programming language places absolutely no restriction on the type of programs that you can write – all that is required of you is that are able to shape your ideas (essentially the wood that we used to build our chair!) into working programs using the commands provided.

### **So why choose AMOS?**

After all, AMOS isn't the only programming language available for the Amiga – you could, if you really wanted to torture yourself, achieve almost exactly the same results using languages such as assembler (machine code), C or even that student's favourite (but not mine!), Pascal. In many respects, these languages offer benefits that make them better equipped to handle certain programming projects – assembler, for example, is considerably faster than AMOS and C is the natural language of the Amiga (most of the Amiga's operating system was written in C), so it's ideally suited to applications programming.

AMOS is such a damned good choice for Amiga programming simply because it offers the sort of low-level programming power normally only associated with assembler, but with the kind of user-friendliness only associated with BASIC, a language designed specifically with beginners in mind (this is hardly surprising when you consider that AMOS is built upon the BASIC language anyway!). Assembler language, on the other hand, is notoriously difficult to learn, especially when you consider that you also need to learn the Amiga's hardware and operating system inside out before you can get even the simplest of programs to work. AMOS shields you from all this by providing a ready-made set of commands that give you with power of assembler but without the hassle! And, because most of the hard work is done for you, AMOS programs are considerably smaller than their assembler equivalents, which means that they don't take so long to write.



**Other  
programming  
languages**

What's more, AMOS also turns in some pretty impressive performance ratings, something that certainly isn't usually associated with most BASIC dialects (as anyone who used Commodore's AmigaBASIC

If you're a complete beginner, then Easy AMOS is for you. Designed more as a teaching tool than a serious alternative to AMOS, Easy AMOS is still capable of great things.



**Easy AMOS**

```

EASY AMOS  Run  Test  Indent  Blocks Menu  Search Menu
           Tutor  Help  Overwrite  Fold/Unfold  Line Insert
I  L:1  C:1  Text:22640  Chip:807224  Fast:7885000  Edit: Hi_Scores.AMOS
-----
HI-SCORE Procedures
By J.P.Cassier and F.Lionet
(c) Europress Software 1992
-----
These procedures are intended to be inserted straight into your own games!
The definitions can be found at the end of this program.

Just highlight the routines you need with the right mouse button, and grab
them into memory with the Block Cut command from the Blocks Menu (Ctrl-C)
You can now save them into a separate file with Block Save, and
Merge them directly into your programs!

Before using these routines in your own programs, you'll need to enter direc
mode, and save off bank 10 as HISCORE.ABK. Then merge the procedures into
your own program using the MERGE command, and then load in HISCORE.ABK
You'll then be ready to go!

Procedure HISCORE_DISPLAY
-----

```

Interpreter will no doubt agree!). Indeed, AMOS is probably as fast as your average C program, even before it has been compiled! In many ways, AMOS is still the next best thing to assembler language.

## AMOS at its best

Ok, time for bit of a 'U' turn – Although I've gone to great lengths to extol AMOS' ability to more than adequately handle just about any programming project thrown at it, it has to be said that AMOS really comes into its own when applied to games programming. Although it is still being used extensively as a vehicle for databases, word processors and education software (Europress's own 'Mini Office' and 'Fun School' products vividly demonstrate this), AMOS was originally designed with the games programmer in mind.

Just take a look at the sort of tasks that AMOS can handle – high speed screen scrolling, interrupt-driven animation, multiple blitter objects and sprites, interrupt driven music and the ability to play sound samples. Come on, let's face it – these are hardly the sort of facilities you'd expect to find in a programming language designed for writing databases! AMOS is a programming language designed with games programming in mind and that's a fact.



**Blitz Basic 2,  
3D  
Construction  
Kit**

Once again, there are alternatives available, but no other games creation system even comes close. OK, so Acid Software's Blitz Basic 2 is getting there, but distribution problems have stopped it from gaining any real foothold in any country other than New Zealand. Domark's 3D Construction Kit certainly gives AMOS 3D a run for its money, but, because it's not a programming language, it places too many restrictions on the user to be of any real use. Once again, this is where AMOS really scores over the competition – think of any particular genre of game, and the chances are that AMOS will be able to handle it.

---

## The AMOS family

The release of AMOS, way back in 1990, spawned a whole range of support products, not just from Europress, but from third party developers too. Compilers, alternative sprite editors, sound manipulation utilities and even a complete 3D graphics extension are all now available to the AMOS programmer, making AMOS the most flexible language available for the Amiga. Even AMOS itself is no longer a single product – in an attempt to make AMOS of interest to all Amiga users, AMOS is now available in three different flavours, each aimed at a particular user.

### 1. Easy AMOS

Many users found the original AMOS somewhat too complicated (which wasn't particularly surprising as the manual was so appalling), so Europress did the decent thing and launched Easy AMOS. Designed more as a teaching tool than a serious rival for AMOS, Easy AMOS offers pretty much the same facilities as its big brother apart from some minor omissions (the AMAL animation language, for example). It's nowhere near as powerful as AMOS Professional, but it still manages to provide an ideal introduction to AMOS programming.

In many ways, Easy AMOS laid the foundations for AMOS Professional, the successor to AMOS 1.35. Many of the features introduced in Easy AMOS eventually found their way across into AMOS Professional including Easy AMOS's excellent Sprite Editor and Sample Bank Maker accessories, and the very useful 'Tutor' facility (this was enhanced and renamed the 'AMOS Monitor' in AMOS Professional). Indeed, much of Easy AMOS was actually superior to AMOS 1.35.

What makes Easy AMOS such an attractive proposition to beginners is its excellent manual which was written by computer industry veteran Mel Croucher, he of AutoMata PieMan fame (oops, I showed my age a bit there). Written more as a step-by-step walk-through of the Easy AMOS editor and language than a traditional reference manual, the Easy AMOS manual is ideally suited to beginners as it not only documents Easy AMOS, but also teaches the fundamentals of computer programming.

## 2. AMOS 1.35

Francois Lionet's original AMOS, version 1.0, was released onto an expectant Amiga market way back in late 1990, almost a year later than expected. For the first time ever the average user could produce programs of the sort of quality that would have turned your average assembler programmer green with envy. Using just a few simple BASIC-like commands, AMOS users were able to scroll screens at high speed, play music modules and sound effects, display IFF pictures and move sprites and blitter objects around the screen at break-neck speeds. No wonder AMOS became such an overnight success!

AMOS has undergone many revisions since that first release, the latest of which – at the time of writing – is version 1.35, a fairly minor revision that makes AMOS fully compatible with Workbench 3.0-based machines. Before that, AMOS was upgraded quite substantially with the release of version 1.34, which not only fixed a few bugs, but also added many new features including direct support for Sound Tracker modules. If you own a copy of AMOS that hasn't already been upgraded, then contact your local PD supplier for the latest AMOS 'updater' disk. Many AMOS extensions and utilities (the AMOS Compiler, for example) will not work on versions of AMOS older than 1.34, so it's up to you to keep your copy of AMOS as current as possible. Better still, why not upgrade to AMOS Professional!



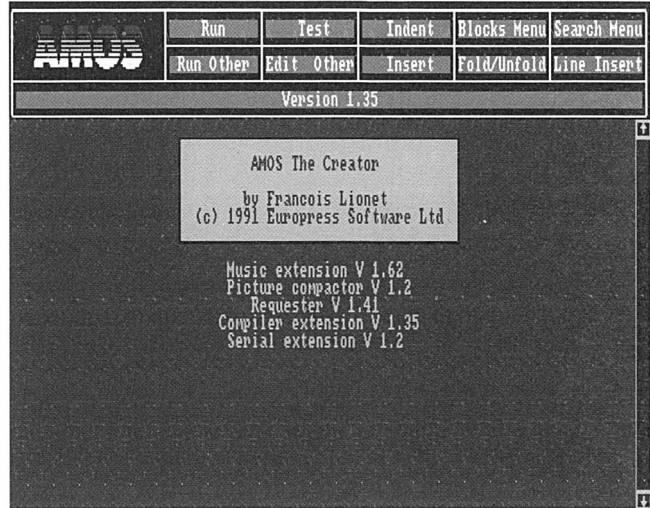
AMOS 1.35 is no longer seen as a commercially viable product by Europress and has, as a result, found its way onto several magazine coverdisks including the January 1993 issue of Amiga Format. If you'd like to test the water before splashing out on something like AMOS Professional, buying a back issue of one of these magazines is a sensible move. Although you don't get a manual with these special magazine



The original AMOS may be showing its age a bit these days, but it's still one of the most powerful BASICs available for the Amiga.



**AMOS 1.35**



versions of AMOS, at least you can follow the examples published within this book and perhaps, once you feel confident enough with AMOS, you'll want to take your AMOS coding further by buying a fully packaged version!

### 3. AMOS Professional

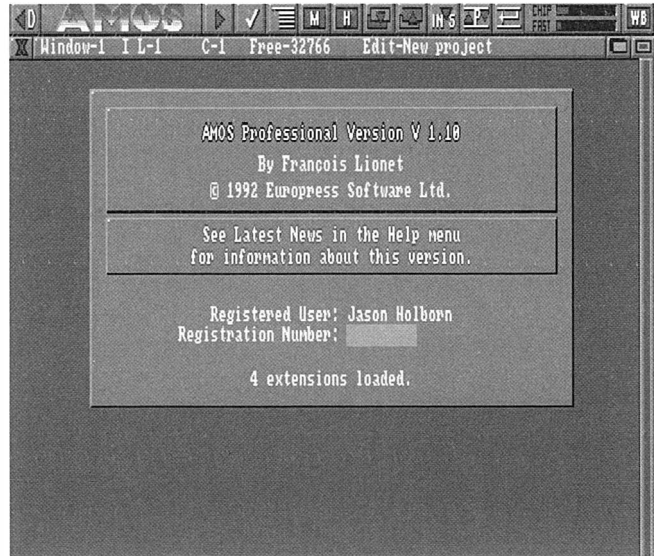
Up until the last few weeks of 1992, AMOS ruled the roost, but all that has now changed with the release of AMOS Professional, the latest and most certainly the greatest version of AMOS yet. Most of the AMOS industry were expecting Europress simply to enhance AMOS 1.35 by bolting on a couple of extra bits and pieces, but Europress have done us all proud by virtually rewriting AMOS from the ground up. Although the foundations have remained pretty much the same, AMOS Professional is far slicker and certainly more capable than its predecessor. If you're serious about your AMOS coding, then this is the one to buy.

Apart from some flashy new packaging and an excellent manual written by Mel Croucher (the author of the very readable Easy AMOS manual), AMOS Professional boasts many new and improved features that make it a definite step up from AMOS 1.35. The most immediate of these is AMOS Professional's editor, which is a vast improvement over the rather quirky editors employed by both AMOS 1.35 and Easy AMOS. Designed to look and perform more like Workbench 2.0, the AMOS Professional

*If you want the best, then AMOS Professional is the one to go for.*



**AMOS  
Professional**



editor lends itself so much better to heavy coding sessions. Cosmetic changes aside, the new editor offers many powerful features including pull-down menus, keyboard macros, multiple windows (even multiple source files) and some of the most advanced undo/redo facilities ever to be found within an Amiga text editor.



**Pro's extra  
features**

The enhancements offered by AMOS Professional go so much further than just the editor, though – it also offers a plethora of new commands (over 200 in total) that extend AMOS to over 700 commands! These include commands that allow AMOS programs to communicate with ARexx ports (and therefore other programs), load and play animations in standard IFF ANIM format (AMOS can actually run animations faster than Deluxe Paint!), perform double precision floating point maths (handy if you own an Amiga equipped with a maths co-processor chip), play Sound Tracker and MED modules and so much more besides.

By far the most important addition to AMOS Professional, though, is its powerful 'Interface' language, an interrupt-driven subset of AMOS commands that have been specifically designed for the task of handling user interfaces (the bit of your program that the user sees on the screen). Interface allows you to quickly and easily create complex front ends for

your programs, complete with buttons, scroll gadgets and active lists. What's more, AMOS Professional does most of the work of handling these gadgets for you – all you have to do is to check each gadget now and then to see whether the user has clicked on any of them.

---

## AMOS Utilities

Although AMOS comes complete with a fairly comprehensive selection of tools to aid you with your programming, both Europress and several third party developers have been quick to produce a range of add-ons that further expand AMOS's already impressive arsenal of features. Read on to find out more...

### **AMOS 3D ● £34.99 Europress Software**

One of the first and arguably the most impressive add-ons ever to be released for AMOS was AMOS 3D, an extension to AMOS specifically designed to handle 3D graphics. Written by 3D specialists Voodoo Software (authors of the hit 3D game Xiphos), AMOS 3D adds over 30 new commands to AMOS that enable you to manipulate 3D objects in real-time to create StarGlider-like effects. Up to 20 3D objects can be placed onto the screen and then moved around, rotated and even stretched in 3D space at speeds that more than match commercial 3D games such as Elite and StarGlider.

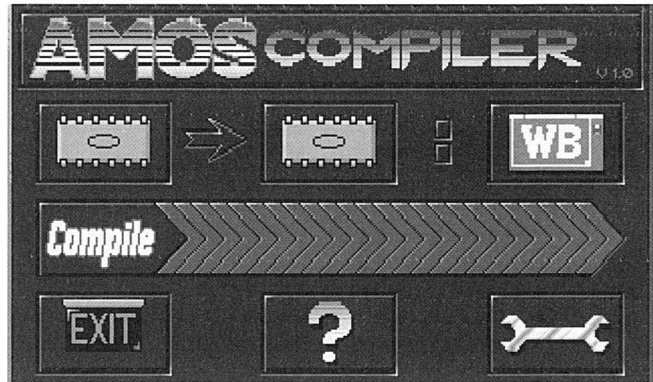
The heart of AMOS 3D is undoubtedly the language extension, but AMOS 3D also comes complete with a very powerful 3D object editor called 'OM' (short for 'Object Modeller') written by Voodoo to handle the task of constructing 3D objects. Working in 3D is usually a rather hair-raising experience, but OM makes the process of designing even the most complex of 3D objects child's play.

### **The AMOS Compiler ● £29.99 Europress Software**

In order to distribute AMOS programs to other users who may not already own the AMOS Interpreter, Europress have produced the AMOS Compiler, which can take any AMOS source program and convert it into stand-alone machine code. Like AMOS 1.35, this too has found its way onto magazine coverdisks, so keep your eyes peeled for back issues.



Add an extra spurt of speed to your AMOS code with the AMOS Compiler.



The AMOS Compiler consists of a very pretty Compiler ‘Shell’ program and an extension to the AMOS language. The extension effectively builds the compiler into AMOS, allowing the Shell to be run as an accessory within AMOS itself. The compiler offers many advantages to the AMOS programmer, including the ability to produce code that is not only independent of AMOS 1.35, but also runs considerably faster than the original source code – Europress quote speed increases of over 60%! What’s more, compiled programs are automatically compressed too, considerably reducing the size of your programs.

### **AMOS Pro Compiler ● £34.95 Europress Software**

Although the standard AMOS 1.35 Compiler will happily compile programs written under AMOS Professional (note that previous versions don’t work with AMOS Pro), only code that is downwardly-compatible with AMOS 1.35 will work. AMOS Pro provides a ‘Check 1.3’ menu option that allows you to check whether your code is compatible with AMOS 1.35. The only disadvantage with this is that the standard AMOS Compiler can’t handle the extra commands provided by AMOS Professional for such things as IFF animation and the powerful Interface language.

Obviously, using the standard AMOS Compiler to compile source code from AMOS Professional defeats the object of owning this enhanced version of AMOS altogether, so Europress have also released the AMOS Pro Compiler, a completely new compiler written specifically to handle not only the standard AMOS instruction set, but all those new AMOS Pro

The AMOS Pro compiler is a considerable improvement over the original AMOS compiler. Even if you don't own AMOS Pro, this is the compiler to buy.



**AMOS Pro  
Compiler**



commands too. The new compiler works with both Easy AMOS and AMOS 1.35, but it really comes into its own when used with AMOS Professional. What's more, you don't even need AMOS to use the AMOS Pro compiler. You can even write your AMOS code using your favourite ASCII text editor (Cygnus Ed, for example) and then compile it directly without ever having to load up AMOS Professional! This will allow you to create a development environment to C by running both the editor and compiler from the Shell.



**Stand-alone  
compiler**

Not only is the AMOS Pro Compiler powerful, it's also very easy to use. Running from a front end similar to the look and feel pioneered by AMOS Pro, the AMOS Pro compiler offers a far more intuitive front end that is a doddle to use. What's more, Europress have designed the compiler to be far more integrated into the AMOS Professional programming environment. What this basically means to the average programmer is that the compiler will look and feel as if it were actually built into AMOS – a bit like the AMOS Professional Sprite Editor.

## **AMOS Tome ● £24.99 Shadow Software**

Shadow Software are a household name to established AMOS users. One of their first releases was AMOS Tome (Total Map Editor), a professional version of the 'TAME' Map Editor bundled with the original AMOS. Tome adds over 60 commands to AMOS that let you create huge game play areas that use up very little memory.

Why is Tome so useful? Well, you've probably already played games that use a very similar technique to that employed by Tome – just check out any game (Team 17's hit 'Assassin' and 'Alien Breed' games are good examples) that allows you to scroll around a play area that appears to be much larger than the tiny display that can be viewed on your Amiga's monitor. In order to save memory, the play area used by these games are built up like a jigsaw using tiny 'tiles' that can be used over and over again. And, because the tile needs only to be held in memory once, you get a considerable saving in memory.

Don't worry if map-based games sound complicated – AMOS Tome provides you with a whole selection of commands specifically designed to take the hard work away from you. All you have to do is to design the play area using the Tome Map Editor, load it into AMOS and then call a few simple commands to get your Alien Breed beater up and running!

### **CTEXT ● £3.50 Shadow Software**

Shadow Software strikes back with CText (short for Colour Text), an AMOS extension that allows you to use fonts with up to 64 colours within your AMOS programs. CText does cheat a little – unlike the ColorText facility built into Workbench 2.0 upwards, the colour fonts that CText uses aren't true fonts. Instead, CText stores its colour fonts as an AMOS Icon Bank with each letter treated as a single icon that can be pasted down anywhere on the screen. In some ways this is a more flexible approach – you can use CText's colour fonts for scrolling messages.

CText does make handling these Icon-based characters easier, though, by providing you with a number of commands designed specifically to handle colour fonts. The most important of these is the 'Ctext' command that is basically the CText equivalent of AMOS's own 'Text' command.

### **D-SAM ● £19.95 AZ Software**

If you need to be able to manipulate sound samples, then AZ Software's D-Sam is for you. D-Sam is an AMOS extension that adds over 46 new commands to AMOS that will allow you to perform all manner of sample-based operations, including the ability to play sound samples direct from hard disk or floppy. By playing your samples from disk, even a 512K Amiga becomes capable of playing samples megabytes in size!

D-Sam also provides direct support for Aegis AudioMaster's sample sequences facility, which embeds a whole series of loop points into a single sample, therefore giving the impression of a much larger sample when played back. D-Sam also provides such powerful sample manipulation facilities as fading, oversampling, playing of raw, IFF and even compressed samples.

### **NCOMMAND ● £7.50 Oasis Software**

If you need to add an Intuition-like front end to your AMOS programs but you're not lucky enough to own AMOS Professional, then Oasis Software's NCommand could prove a worthwhile alternative. Although not strictly an AMOS extension, NCommand consists of a series of powerful procedures that can be merged into your code and then called to create Workbench 2.0-like user interfaces complete with scroll gadgets, buttons and even the new 'rollo' gadgets and 'radio' buttons added to Intuition with the release of Workbench 2.04.

---

## **AMOS futures**

Europress isn't the sort of company that rests on its laurels, so it should come as no surprise that the AMOS family is expanding even now. Here's a quick rundown of the products that Europress hadn't released at the time of writing. By the time you read this book, though, they should hopefully be available, so keep your eyes peeled...

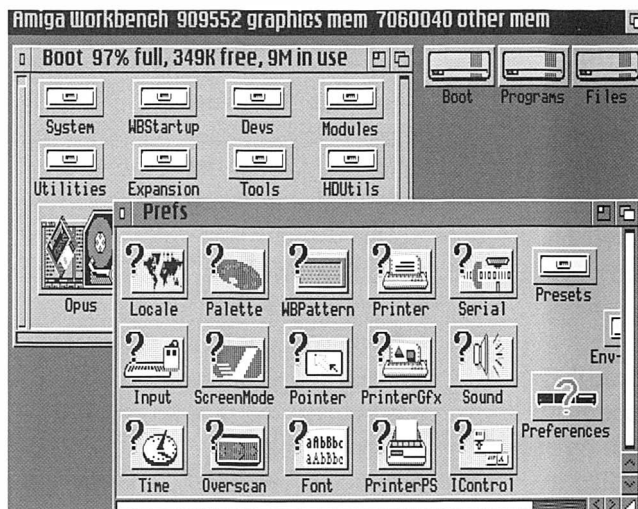
### **Intuition Extension**

We all know that AMOS is the 'bee's knees' when it comes to programming the Amiga's hardware-level facilities such as scrolling, sprites and bobs etc, but there is one area of AMOS that is sadly lacking – Intuition support. Intuition is the name given to the Amiga's WIMP management system, the part of the machine's operating system that handles all those pretty windows, pull-down menus and gadgets that you see on the Amiga's Workbench.

Many would-be applications programmers have turned their noses up at AMOS in the past, simply because it is not possible to write programs that communicate with the user through Intuition windows and gadgets. In some ways, this view is quite understandable – after all, the whole



The forthcoming AMOS Intuition extension will let AMOS Professional programmers write applications that run under Workbench, making AMOS ideal for applications programming.



point of the Amiga is that it can multitask and (whilst AMOS can multitask to a certain extent), you really do need to run applications under Intuition to get the full benefits of multitasking.

Help should be at hand very soon, though, as Europress plan to release an extension to AMOS Professional that will allow applications programmers to open screens and windows under Intuition. Coupled with AMOS Professional's powerful 'Interface' language, this extension could prove to be the key to AMOS's final domination of the Amiga programming language market. Let's hope so!

## AGA support

By the time you read this book, Europress should hopefully have shipped the long-awaited AGA-compatible release of AMOS Professional. Obviously this won't make a lot of difference if you're running AMOS Professional on an Amiga based around the original or enhanced chip sets (the A500, A600, A1500 etc), but it's well worth acquiring if you're one of those people with enough charm to sweet-talk your parent/spouse into forking out for an A1200 or A4000 (my girlfriend wouldn't let me have an A4000 until this book had been written!).

Specific details were a little thin on the ground at the time of writing, but Europress's project manager (a very nice chap who goes by the name of

Richard Vanner) assured me that AGA-support will enable AMOS programmers (that's you and I!) to write AMOS programs that take full advantage of the vastly enhanced graphic capabilities offered by this super-doooper chip set. This will probably mean that you'll be able to access the extended colour palette offered by AGA (don't forget that the AGA chip set offers a full 24-bit colour palette – that's over 16.7 million colours!) and hopefully also the new 256-colour and HAM-8 screen modes.

We won't actually be covering either of these new screen modes within this book, simply because Europress were still hard at work on the upgrade at the time of writing, but I will try to include as much AGA information as possible. My guess is that the 'Screen Open' command, Rainbow commands and the colour palette will probably be the only aspect of AMOS to feel any AGA influence.

---

## Getting started with AMOS

So you're already the proud owner of an Amiga and you've taken the plunge and bought yourself a copy of AMOS (be it Easy AMOS, AMOS 1.35 or AMOS Professional, it doesn't really matter – all three of them are very powerful). All that now remains is to get stuck into some coding, right? Wrong! Although an Amiga running AMOS will allow you to create a fairly wide selection of programs, some extra bits and pieces are highly recommended if you want to take AMOS to the max.

Let's start with a look at the sort of hardware setup that you'll need. Well, obviously you'll need an Amiga – what type of Amiga you own doesn't really matter (AMOS doesn't discriminate), but some additional memory is well worth investing in. All three versions of AMOS insist on at least 1Mb of RAM, but – to be perfectly honest – this isn't really enough if you want to create anything more than the simplest of programs. Games and demos take up lots of memory (especially if you start to use long sound samples and lots of colourful graphics), so an extra 1 Mb of RAM should be at the top of your shopping list.



### Hardware requirements

A hard drive definitely makes AMOS a much friendlier beast to work with (especially AMOS Professional), but don't feel that your system is

second best if you can't afford such a luxury (a twin drive system is a more than acceptable alternative). With the release of machines such as the IDE-equipped A600 and A1200, hard drives have dropped in price faster than stock market shares, so you might just be surprised just how cheap hard drives have become. If you do decide to take the plunge, you'll be amazed just how much a hard drive enhances AMOS – not only do your source files load faster, but the AMOS accessories are permanently on tap and everything runs so much smoother.

A monitor is a definite must if you intend to code anything more than the odd little utility. Because you'll be doing a lot of typing, trying to run AMOS on a TV could eventually damage your eyes. Televisions are OK for running games and watching demo programs, but programming is a totally different kettle of fish altogether – if you can afford a monitor, then do your eyes a favour and get one. Even a cheap green-screen monitor will do for programming. You can then swap back to your TV as soon as you need colour.

Finally, we come to the subject of software. Although Europress have done their very best to provide you with all the tools that you will need to use AMOS fully, there are still a couple of extra programs that you'll need. Don't worry, you don't need to spend out a lot more money – most of these you'll probably already have anyway. Even if you don't, most of the programs mentioned below can be picked up for little more than peanuts these days.



**Extra  
software**

## **1. Get a paint program**

Even if you've only just bought your Amiga, the chances are you already own a paint program such as Electronic Arts' excellent Deluxe Paint. Even though the AMOS Sprite Editors are pretty capable (particularly AMOS Professional's Sprite Editor), it's still worth having a paint package handy for drawing backgrounds, title pages and even sprites.

Most AMOS programmers tend to use DPaint for designing sprites that are then pulled across into the AMOS Sprite Editor. This is certainly how I work – although I'm a great fan of the AMOS Sprite Editor, nothing can touch DPaint for its speed, convenience and power.

The AMOS Sprite Editor is a very powerful beast indeed, but you'll still need a decent paint package for background graphics, title screens and even some sprites.



**DPaint**



## 2. Get a sound sampler

If you intend to write games using AMOS, a sound sampler is a definite must. Even if your game is the most playable thing since Rainbow Islands, it's an unfortunate fact of life that game players often judge even the most playable of games on the quality of their graphics and sound. DPaint pretty much covers the graphics aspect, but your game won't be half as impressive if you use the AMOS 'Shoot' and 'Bang' commands to generate your game's sound effects.



**StereoMaster  
AMAS 2**

With a sound sampler such as MicroDeal's excellent StereoMaster or AMAS 2 (my personal favourite!), you'll be able to grab all manner of weird and wonderful sound effects from just about any audio source.

## 3. Get a sound tracker

Unless you've already experienced the delights of Public Domain software, the chances are that you haven't encountered the Sound Tracker utility. Originally designed by hackers for writing sound tracks for demo programs, Sound Trackers are now so popular amongst programmers that even most commercial games programmers use them these days! Based around an editing system similar to a drum machine, a Sound Tracker allows you to write music by arranging sampled instruments into short 'patterns' which are then strung together to form songs.



AMOS provides direct support for both the standard Sound Tracker 'MOD' format (short for module) and Teijo Kinnunen's excellent Sound Tracker alternative, MED. If you want to write your own music for your AMOS programs, then you should strongly consider spending a couple of quid on a PD disk that contains a decent Sound Tracker. Coupled with a sound sampler (as discussed above), you could even write tunes complete with your own instrument samples!

#### **4. Put the kettle on!**

Any type of programming is thirsty work, so always keep the kettle topped up and a ready supply of coffee and chocolate chip cookies to hand. Music is always a good bet too if you want the old creative juices to flow – what you choose is up to you, but I personally favour a bit of Depeche Mode's 'Songs of Faith and Devotion'...

---

## **Selling your Software**

If you've written an AMOS program that you're particularly proud of, you may want to distribute it so that other AMOS users can get their hands on it. This sort of thing is a good idea as it will not only allow other Amiga users to benefit from your programming prowess, but – providing you handle it correctly – you may even get a little fame and fortune chucked in for good measure. Unlike some game creation packages I could mention, it's perfectly possible to actually sell AMOS programs. If yours is extremely good, you could even have it marketed as a commercial product which could (hopefully) make you a lot of money.

AMOS is such a powerful programming language that there's absolutely no reason whatsoever why it couldn't be used to produce software of commercial quality. Quite a few titles have already made it – Europress's Mini Office and even its range of Fun School education titles being just two examples – so there's no reason why you too couldn't get in on the act. Obviously, software houses such as Europress don't accept any old tosh – your program (be it a game, utility or application) needs to be of a suitable standard to be of marketable value.

We don't really have the space here to discuss the various options available to you, but by the time you reach the end of this book

(providing you read it from cover to cover), you too should be capable of producing AMOS programs of commercial quality! Oh, and if you do become rich and famous, don't forget me will you – I'll be happy to accept an Escort RS Cosworth as a token of your gratitude!



# How AMOS works

- The AMOS Editors (Easy AMOS, AMOS 1.35, AMOS Pro)
- The AMOS Monitor (Easy AMOS and AMOS pro only)

**P**rogramming in AMOS may come as bit of a shock if you're more used to the sort of unfriendly programming languages that the so-called 'professionals' swear by. Unless you enjoy spending the first few days of your time with a programming language actually installing the damned thing onto ten floppy disks and then another few days trying to figure out how to run the damned thing, I'm afraid AMOS is going to come as bit of a disappointment. There's no complicated setup procedure (well, Easy AMOS does need to be installed, but even then all the work is done for you), no complex commands to type in just to get AMOS running – simply insert your AMOS program disk and AMOS will spring to life.

Once AMOS has loaded, you'll be presented with a very pretty-looking AMOS Editor. How your particular editor looks depends entirely upon what version of AMOS you're running – Easy AMOS owners have to contend with a cyan on dark blue front end, whilst AMOS 1.35 owners have the same cyan on blue but there's also some very nice red and white in there too. As for AMOS Pro owners, well you lot have the ultimate in AMOS editors!

AMOS is perhaps the first true example of a totally integrated programming environment. Although such a tag sounds rather complicated, what it essentially means is that every single step involved in the development of a program can be carried out from within the AMOS Editor. Whether you're writing the program's code, designing sprites or pulling together all the sound samples that your program uses (the last two of these tasks can be carried out using AMOS's powerful accessory programs), you need never leave the AMOS Editor. Obviously there are some tasks that still need to be carried out within a separate program – writing a game's soundtrack, for example – but even then the wonders of the Amiga's multitasking operating system allow other programs to be run concurrently alongside the AMOS editor. Isn't the Amiga wonderful!



### **'Integrated' programming**

Over the next few pages or so, we'll be taking a pretty in-depth look at the editors used by the three different versions of AMOS. The Easy AMOS and AMOS 1.35 editors are virtually identical, so anything that you read that applies to AMOS 1.35 will almost certainly apply to Easy

AMOS too. AMOS Pro offers a number of extra editing options that aren't present in other versions, so I've tried to mention these too.

---

## The AMOS Editor

The AMOS Editor is essentially a text editor specifically designed for handling AMOS source code files (your programs). It allows you to type in and edit AMOS programs, load and save AMOS programs and even run them. This last facility gives away the major difference between the AMOS Editor and a conventional text editor such as ASDG's excellent CygnusEd 2.0 – unlike other text editors, the AMOS Editor has the AMOS language actually built into it. What this essentially means is that you can run your programs from within the AMOS Editor without having to use a separate runtime tool or compiler.



**'Edit' mode**

The AMOS Editor offers two different working modes — Edit and Direct mode. When you first load AMOS, it automatically goes straight into Edit mode, the part of AMOS that lets you actually edit AMOS programs. From here you can type in your AMOS programs, save them off to disk, load existing source code and perform a whole host of editing functions on your code. Through either pull-down menus or an options-within-an-icon-strip (depending upon what version of AMOS you're running), you can also run your source code.



**'Direct' mode**

The second mode on offer is 'Direct' mode which, as its name suggests, provides the AMOS programmer with direct access to the AMOS interpreter. Unlike the programs that you type into the AMOS editor, entering an AMOS command in Direct mode will force it to be performed the moment you press the 'Return' key. Direct mode is a bit like the Shell interface (or 'CLI', if you like) offered by the Amiga's Workbench. Every command you type is executed immediately, allowing you to carry out tasks such as loading graphic files without having to code them directly into your programs. (The reasons for doing this will become more apparent later when we look at AMOS memory banks.)

You can toggle backwards and forwards between Direct mode and Edit mode by pressing the 'Escape' key on your Amiga's keyboard.

## Editor options

Running along the top of the Easy AMOS and AMOS Editor screens are a strip of icons that contain a selection of editing functions that help to make working with the AMOS Editor that bit more productive. Just like a conventional text editor, the AMOS Editor includes a whole host of editing options including full block cut, copy and paste, ASCII merge and the usual search and replace options. Not all of these options are immediately accessible, though – because only ten options can be displayed at once (there are only ten icons), most of them are hidden from view.




### Accessing functions

In order to access all these extra functions, you'll need to hold down one of three 'qualifier' keys – Control, Shift or ALT. To actually select an option, simply move the mouse pointer over the icon you want and then press the left mouse button. Alternatively, you can avoid having to reach for your digital rodent altogether and simply press one of the Amiga's ten function keys. Each function key corresponds directly with each of the ten menu icons on offer – F1 will select the top left icon, F2 will select the icon immediately to the right and so on. It's a little hard to explain in words, so I suggest you try it for yourself – you'll be able to see which icon is selected by which function key as AMOS highlights the function that has been selected. Anyway, enough of the theory – let's take a look at what all those menu options actually do. First, the Editor menu...

## The Editor menu

At the very top level of AMOS' menus is the Editor menu that provides all those important options required to get your code up and running once it has been entered. No qualifier keys need to be held down to access this menu — it's permanently on tap, so simply move the mouse pointer over the icon you want and click on it.

	Run	Test	Indent	Blocks Menu	Search Menu
	Run Other	Edit Other	Insert	Fold/Unfold	Line Insert
Version 1.35					

**Run**

**RUN** Typing in a program is all very well and good, but it won't actually do a lot unless you select this option (or press F1). When you do run your program, AMOS first checks through it to make sure that there are no syntax errors (you've entered a command incorrectly, for example). If everything checked out fine, AMOS then runs your program.

**Test**

**TEST** Testing a program may seem rather pointless, but it's very important if you wish to compile your code at a later date. Testing your code forces AMOS to check through each and every line for any syntax errors that may have sneaked in. You should note, however, that AMOS can't spot logic errors (that is, errors that will cause your program to do something that it shouldn't – a sprite moving in the wrong direction, for example). A syntax error can be anything from an incorrectly spelt command (typing PRNT instead of PRINT) or the incorrect use of a command (A\$=Print, for example).

**Indent**

**INDENT** The Indent option comes in very handy if your code is somewhat unreadable. What it does is to tidy up the appearance of your program by indenting code within loops and control structures. If you don't already use this option (or, like me, intend code automatically anyway), try it – I think you'll be surprised just how much more readable your code will become.

**Blocks Menu**

**BLOCKS MENU** Clicking on this option will take you into the blocks menu. It's basically the same as pressing the Control (CTRL) key.

**Search Menu**

**SEARCH MENU** Another quickie way of accessing another bank of ten icons. The Search menu provides you with a selection of options for locating text within your code.

**Run Other**

**RUN OTHER** The AMOS Editor is capable of holding more than one program in memory at once by loading each program in individually using the 'AC.NEW/LOAD' option from within the System menu (don't worry, we'll get to this menu shortly). Selecting this icon will bring up a requester listing all the 'Other' programs currently held in memory. Simply click on the one you want and you're away.



**Edit Other** **EDIT OTHER** The Edit Other option is very similar to the Run Other option but for one major difference – instead of running the program of your choice, Edit Other allows you to edit the program of your choice.

**Insert** **INSERT** This option toggles between the two editing modes offered by the AMOS Editor. By default, the editor runs in ‘Insert’ mode (text is inserted in between any existing characters) but it can also run in ‘Overwrite’ mode (text is typed over existing characters).

**Fold/Unfold** **FOLD/UNFOLD** If you use procedures within your AMOS code, clicking on this icon will cause the procedure that the cursor is currently positioned over to fold (only the procedure’s name is displayed). To unfold a procedure (reveal all the code held within it), simply position the cursor over the procedure name and select this option again. Don’t worry if you don’t understand procedures at the moment – we’ll be covering them in chapter 4.

**Line Insert** **LINE INSERT** Clicking on this option will cause a blank line to be inserted between the current line (the line that the cursor is on) and the line directly above it. You can achieve the same effect simply by pressing the ‘Return’ key.

## The System menu

The System menu provides you with a selection of file management options that give you the option to load and save AMOS programs, load accessories etc. To access this menu hold down either the ‘Shift’ key or the right mouse button.



**Load** **LOAD** Unless you intend to enter a program from scratch, you’ll need to use the Load option to pull in a previously saved AMOS program. Clicking on this option will bring up the AMOS file requester.

**Save**

**SAVE** If you want to keep your AMOS programs for posterity, then click on the 'Save' option. The Save option stores the current AMOS program that you're editing on disk. To make life easier when reloading AMOS programs, always save your AMOS programs with the file extension '.AMOS' (SCRABBLE.AMOS, for example).

**Save as**

**SAVE AS** Unlike the 'Save' option, the 'Save As' option will always ask you what filename you wish to save the current program under. If you click on 'Save' once a program has already been named, AMOS will save the program to disk without even asking you whether you're happy with its filename.

**Merge**

**MERGE** As its name suggests, the 'Merge' option allows you to combine the current program being edited with an AMOS program held on disk. The code loaded from disk will automatically be inserted at the current cursor position.

**Merge Ascii**

**MERGE ASCII** AMOS doesn't store programs in ASCII format, so the standard 'Merge' option won't work if you want to pull in a section of ASCII text into your program. Not surprisingly, this option will do the job instead.

**AC.New/Load**

**AC.NEW/LOAD** AMOS allows you to load and run accessories (the AMOS Sprite Editor, for example) directly into the AMOS Editor, allowing you to edit sprites, generate AMAL code etc without having to save your AMOS code first. The AC.NEW/LOAD option clears any accessories that may have already been loaded and then loads all files that end in '.ACC' into memory. Once loaded, use the 'Run Other' option from the Editor menu to actually run an accessory.

**Load others**

**LOAD OTHERS** If you can't afford the memory to hold a whole set of accessories in memory at once, then use the 'Load Others' option to load a single accessory into memory.

**New Others**

**NEW OTHERS** If memory starts to get a little tight, then you can delete one or all accessories using this option.

**New**

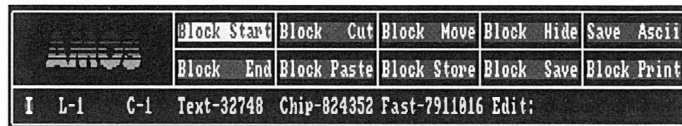
**NEW** Need to wipe the slate clean and start coding again from scratch? Click on this option and the current AMOS program being edited will be wiped from your computer's memory, along with any banks that it may have assigned. Just to make sure you haven't totally flipped, AMOS will give you the option of saving your code first.

**Quit**

**QUIT** In the unlikely event that you've had enough of coding for one day, selecting the 'Quit' option will exit AMOS and return your Amiga to the Workbench or CLI that AMOS was originally launched from. Note that you don't have to exit AMOS to access the Workbench or CLI – just press the left Amiga key and 'A' to toggle between AMOS and the Workbench.

## The Blocks menu

The Blocks menu provides the AMOS programmer with a healthy selection of editing tools that allow you to cut, copy and paste whole sections of code. Simply hold down the 'Control' key to access this menu.



**Block Start** **BLOCK START** In order to mark a block, you must tell AMOS where it starts and finishes. Position the cursor where the block is to start and then click on this option to mark the Block's start position. Alternatively, press the right mouse button and drag to mark the block.

**Block Cut** **BLOCK CUT** Once a block has been defined, it can be removed by selecting this option. Note that once a block has been cut, it can still be rescued simply by clicking on the 'Block Paste' option.

**Block Move** **BLOCK MOVE** The 'Block Move' option comes in handy when you need to move a block of code from one place to another within your program. Simply mark the block to be moved, position the cursor where the block is to be moved to and then click on this option.

**Block Hide** **BLOCK HIDE** If you no longer need a block that you've defined, click on this option and the block will be deselected.

**Save Ascii** **SAVE ASCII** The 'Save ASCII' option saves the current block to disk as an ASCII file. Note that it should be stored first using the 'Block Store' option.

**Block End** **BLOCK END** Once you've marked the start of a block, you need to mark the end of the block using this option. Alternatively, just let go of the right mouse button.

**Block Paste** **BLOCK PASTE** Once a block has been stored or cut, it can be pasted down anywhere within your program using this option. Although very similar to the 'Block Move' option, 'Block Paste' can paste down the same block as many times as you like.

**Block Store** **BLOCK STORE** Block Store takes a copy of the currently defined block and effectively 'remembers' it. Once it has been remembered, the block can be pasted, saved or printed.

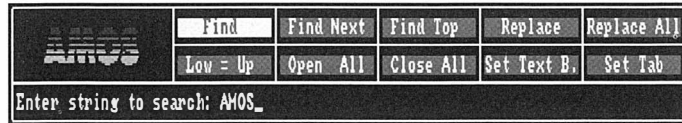
**Block Save** **BLOCK SAVE** This option is very similar to the 'Save ASCII' option, but instead of saving the file in ASCII format, the 'Block Save' option saves the current block to disk in AMOS format which can later be loaded or directly merged into another program.

**Block Print** **BLOCK PRINT** AMOS does provide a direct method of getting a printout of a program listing, so the 'Block Print' option comes in handy. Simply define the block you wish to print, store it (using the 'Block Store' option) and then click on this little beauty. Providing that your printer is all set up and ready to go, you should get a printout of the currently defined block.

## The Search menu

The Search menu comes in very handy when you need to locate a particular string of text within a listing. Instead of you having to manually search through the listing yourself, AMOS will happily do the job for you and even replace any instances that it finds with an alternative of your choice. You could, for example, change the name of a

variable simply by specifying the name of the variable and the name it is to be changed to. You don't have to search for this menu though – just hold down the 'ALT' key!



### Find

**FIND** AMOS's 'Find' option allows you to search for every occurrence of a given string within your program. Note that AMOS will not search folded procedures – these should be unfolded before you start the search.

### Find Next

**FIND NEXT** Once AMOS has found the first occurrence of a string, select this function to move onto the next occurrence.

### Find Top

**FIND TOP** The 'Find Top' option is identical to the 'Find' function, but instead of starting the search from the current cursor position, 'Find Top' starts from the first line of your listing.

### Replace

**REPLACE** The 'Replace' option allows you to replace one string with another. Note that this function only changes the first occurrence, so you'll need to restart it once a string has been replaced. Better still, use the 'Replace All' explained below...

### Replace All

**REPLACE ALL** The 'Replace All' option does essentially the same job as the 'Replace' option but instead of changing just the first occurrence, 'Replace All' replaces every occurrence that it finds.

### Low = Up

**LOW = UP** This option acts as a toggle to control the case-sensitivity of the searching algorithm. By default, this option is set to 'Low = Up' (case insensitive) but it can also be changed to 'Low <> Up' which turns on case-sensitivity (i.e. 'HAPPY' is different from 'HaPPY').

### Open All

**OPEN ALL** Because folded procedures are not searched, the 'Open All' option can be used to unfold every procedure within your program, therefore automatically including the code within those procedures in any search operations you carry out.

**Close All**

**CLOSE ALL** The opposite to the ‘Open All’ option is, not surprisingly, the ‘Close All’ function, which folds all the procedures that are defined within your program.

**Set Text B.**

**SET TEXT B.** This function is used to adjust the size of AMOS’s text buffer (the area of memory used to hold program listings). The larger this setting, the more lines of code can be fitted into memory. Don’t adjust this unless you have to though – the more memory you allocate to AMOS’s text buffer, the less you have left for graphics, music etc.

**Set Tab**

**SET TAB** Not surprisingly, the ‘Set Tab’ function is used to set how many spaces are inserted when you press the ‘Tab’ key on your keyboard.

---

## AMOS Pro menus

Although AMOS Professional no longer uses the same embedded menu system as its predecessors, all of the options covered above (plus a lot more besides) can be found in AMOS Pro’s pull down menus. I’m sure I don’t need to explain how to access these menus – after all, if you’ve already used any other Amiga application that uses pull-down menus, then you’ll already know that all you have to do is to hold down the left mouse button to reveal the menu strip.

AMOS Professional does have its own menu icons (they’re the little squares running along the top of the editor screen), although they are somewhat harder to recognise than their AMOS 1.35 and Easy AMOS counterparts (let’s face it, Europress couldn’t have made the originals any plainer!). Let’s take a look at what they do. Don’t expect too much detail, though – most of them are virtually identical to their AMOS counterparts, so just refer back to the descriptions above for more information.



**DIRECT MODE** The ‘Direct Mode’ icon switches AMOS back to Direct Mode. Pressing the ‘Escape’ key on your keyboard will have the same effect.



**RUN** Not surprisingly, this is the same as the ‘Run’ menu option in AMOS 1.35 and Easy AMOS. Click on this icon to execute your AMOS program. Once again, pressing function key 1 will do the same job.



**TEST** The ‘Test’ icon checks through your code for any syntax errors.



**INDENT** The ‘Indent’ icon attempts to make your code more readable by indenting code held within loops and program control structures.



**MONITOR** The ‘Monitor’ icon is a completely new function that is only offered by AMOS Professional (Easy AMOS owners have a similar function called ‘Tutor’ but it’s nowhere near as powerful). The AMOS Pro Monitor is documented fully at the end of this chapter, but – for those of you who are interested – it’s essentially a tool designed to make the task of removing ‘bugs’ from AMOS programs as easy as possible.



**HELP** The ‘Help’ icon brings up AMOS Professional’s ‘on line’ help facility. We’ll be covering the Help facility later within this chapter.



**SEND TO BACK** Working with several listing windows at once can be confusing, so the AMOS Pro Editor offers two options that allow you to arrange windows to suit your particular needs. The first of these is the ‘Send To Back’ option that, not surprisingly, sends the currently active window behind any others that are currently open.



**BRING TO FRONT** Another window arrangement function is ‘Bring To Back’ that is used to bring the currently active window to the front of all others.



**EDIT MODE** The ‘Edit Mode’ icon switches the edit mode between insert and overwrite.



**FOLD/UNFOLD** Exactly the same as its AMOS counterpart, the ‘Fold/Unfold’ icon opens and closes procedures. Once again, pressing ‘F9’ will have the same effect.



**INSERT** The ‘Insert’ function inserts a blank line at the cursor position. Exciting, eh?



**MEMORY** These two bars graphically display the total amount of chip and fast memory available to your AMOS program.



**WB** The ‘WB’ icon switches AMOS Professional back to the Amiga’s Workbench screen, allowing you to run other programs simultaneously. If you need to then switch back to AMOS Professional, just press the ‘Left Amiga’ and ‘A’ keys.

## Keyboard shortcuts

If you’re not overly keen on the idea of having to reach for your mouse every time you wish to access a menu function, most of AMOS’s menu functions can also be performed through keyboard shortcuts – that is, by pressing a special combination of keys together. We’ve already discussed how to access the menu function within Easy AMOS and AMOS 1.35 – simply press the correct qualifier key followed by one of the ten function keys. AMOS Pro users have an almost bewildering selection of keyboard short cuts on offer. These are listed along with the option within the editor’s pull down menus, so it may be worth noting down the shortcuts required for any option that you use regularly.



**Quicker  
working**

## A helping hand...

In order to make AMOS Professional as easy to get to grips with as possible, those clever chaps at Europress built in a very handy ‘on line’ help facility that allows you to get helpful information on any aspect of AMOS Professional without having to resort to the manual (not that you’d need the manual now you’ve got this book!).

The AMOS Pro help facility works in two ways. At its simplest level, you can simply call it up by pressing the ‘Help’ key on your Amiga keyboard. This will bring up a window containing a menu of all the subjects covered. Obviously, not every function is listed here simply because of the bewildering number of functions covered. Like the users’ manual, the help facility breaks everything up into more manageable sub-menus. For example, if you want to find out information on the



'Dual Playfield' command, you would first enter the 'Screen Control' menu and then the 'Setting Up Screens' menu.



**Quick 'help'**

A much quicker way to access the information you need on AMOS commands is to simply move the cursor over the first character of the command in question and then press 'Help' – AMOS Professional will then display help information on that command without you having to faff around with all those help menus.

Another handy feature of the Help facility is its ability to automatically load an example program that demonstrates the command in question in action. When you pull up help information on a command, the command will be highlighted (red writing on a black background) in the top left hand corner of the help window. Click on this and AMOS Professional will prompt you whether you want to load the demonstration. If you accept, AMOS will load it into a separate window (you don't even have to save your own listing first!).

---

## AMOS Monitor

Regardless of your programming talent, it's an accepted fact of life that computer programs never work perfectly first time. Even the most experienced programmers make mistakes which often result in their code not working quite how it should. The side-effects of these so-called 'bugs' can be anything from a sprite not being displayed or a sample not playing on cue to a complete system crash! What's more, bugs can be very hard to track down, often resulting in hours (and possibly even days) struggling through printed listings.

Bugs come in many different flavours, too, ranging from old favourites such as the mis-spelt variable, to those really infuriating logic errors that you really have to sweat over for days to resolve. To make the task of hunting down bugs that bit easier, both AMOS Professional and Easy AMOS (not AMOS 1.35, though) have a debugging tool called the AMOS monitor (just to make life complicated, Easy AMOS's monitor is called 'The Tutor'). For those of you who have never come across a program of this kind before, a debugger is a program specifically written to aid the process of hunting down bugs within a program. A debugger

won't remove bugs for you, though, so don't expect to be able to feed your buggy source code in one end and get 'bug'-free source code out the other end – the monitor is simply there help you track down bugs that bit faster.



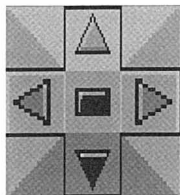
**Monitor**

The AMOS monitor is really quite a simple little tool, but it's very, very useful indeed (as you will find out yourself once you start to use it). What it does is to allow you to examine the inner workings of any AMOS program whilst it is running, therefore allowing you to see at a glance when problems start to occur. You can check the value held within a variable and even the result returned by an expression at any point within a program, giving you the chance to spot bugs far more easily (and therefore quicker). The monitor is instantly available from the AMOS Professional or Easy AMOS Editors – click on the 'M' icon from within AMOS Pro or select 'Tutor' from the menus within Easy AMOS.

You should see a screen appear that is split into essentially four different sections, one of which (the middle one) contains your program listing. At the top left hand corner of the monitor screen is the Graphic Output window that is used to show you a representation of the output from your program. Next to this are a set of icons that control the workings of the monitor (these are explained below).

Below this is the program listing window that, not surprisingly, contains your program listing. As you run through your program, the monitor will automatically move a small pointer through your code to display what instruction is currently being executed, therefore allowing you to see what instructions are causing the results that you see within the graphic output window. Finally, we have the information window at the very bottom of the monitor screen that displays all sorts of useful information as you step through your code, such as error messages any other information you might request.

Anyway, let's take a look at what those gadgets in the top right hand corner of the screen actually do...



**SCREEN SELECT** If the program that you're monitoring opens more than one screen, then you can display any one by flicking through those available using these screen selection controls. Note that only low resolution screens are scaled – if you open a high resolution screen, it cannot be scaled and therefore only a small section of the screen will be displayed.



**INITIALISE** Before you can monitor a program, it must first be initialised. Initialising instructs the monitor to set up the display and to test your AMOS code for syntax errors.



**QUIT** No prizes for guessing what this feature does! Yep, it orders you a pizza... er,... quits you back to the AMOS Pro Editor.



**HELP** If, whilst flicking through a program, you spot an instruction you don't quite understand, simply click on this gadget and then click on the instruction that's baffling you and AMOS Pro will give you a complete break down of the instruction's workings.



**SET BREAK** This function allows you to create what AMOS calls a 'break point'. That is, a position within your program where AMOS will stop the program's execution. Simply click on this icon, click on the command that the break point is to be attached to and voila! — Run the code again and the monitor will always stop at this point.



**EVALUATE** The 'Evaluate' function is very handy indeed. What it does is to allow you to evaluate the result of an expression within your program. For example, if you had an expression that calculated the result of two variables being added together, this option will allow you to instantly see what the result would be at any time during program execution. To use it, simply click on this icon and then click on the first character within the expression and (whilst holding down the left mouse button) drag out a highlight that completely covers all the characters within the expression.



**STOP** The first of the program control icons is the ‘Stop’ icon, which, er... stops program execution.



**SINGLE STEP** If you need to step through your code very precisely, then there’s nothing more precise than single step. Click on this icon and the next single instruction will be executed.



**SLOW** If ‘Single Step’ sends you to sleep, then ‘Slow’ mode may be somewhat more useful. Slow mode continuously executes your program code very slowly indeed, allowing you to see what instruction is being executed at any one time.



**NORMAL** Normal mode runs your AMOS program at full speed in the screen preview display in the top left-hand corner of the monitor screen. Note that the monitor does not show which instructions are being executed.



**FAST** Full speed mode runs at the same speed as normal but allows your program to take over the entire screen, therefore blanking the monitor display. You can still return to the monitor at anytime though, simply by pressing the ‘Control’ and ‘C’ keys together.

---

## Wot, no editor?



Using other  
‘editors’

With the release of the AMOS Professional Compiler, you no longer have to use the AMOS Editor if you don’t want to. Because the AMOS Pro Compiler can accept code that has been saved in ASCII format, you can write your AMOS code using any standard ASCII text editor. Some programmers, especially those more used to languages such as C and Pascal, prefer this way of working as it allows you to work from the Amiga’s powerful Shell (CLI) environment using their favourite text editor (CygnusEd, for example). But this does have its disadvantages.

For starters, an ASCII text editor cannot handle the special permanent memory banks that AMOS can store as part of a program’s source code. These memory banks provide a very convenient method of storing

graphics, music and sprites without having to store them as separate files. What's more, an ASCII editor will not be able to handle the AMOS accessory programs unless they have been compiled first. Even then, the accessories will not be able to automatically pull in and edit data held within a program's memory banks (a very handy facility when editing sprites and icons).

# Programming principles

- Program planning
- 'Pseudo code'
- Sub-routines
- Code comments
- Code indentation
- Procedures
- Handling variables

It can be a lot of fun just experimenting with AMOS. Indeed, I strongly recommend it if you don't fully understand the AMOS instruction set and how the various commands work together. Many of the more complex aspects of AMOS will be covered in quite some depth within this book, so it may be worth your while just tinkering around with sprites, bobs, scrolling etc just to build a solid understanding of these fundamentals within your mind.



**Learning  
BASIC**

Everyone had to start programming at one time or another – yes, even celebrated programmer Jez San didn't know a bit from a byte before he started to program. If you're completely new to this programming lark, then may I suggest that you lay your hands on a decent book on BASIC programming. The 'core' of AMOS is based around a very powerful BASIC interpreter which is almost identical to every other dialect of BASIC available for the Amiga and other computers. You don't even have to buy a book that even mentions the Amiga, let alone AMOS. BASIC is usually so generic that any book on the subject will do the job.

I'm not trying to fob you off here – BASIC programming is such a complex subject that it would fill an entire book on its own!

Meanwhile, the aim of this book is to concentrate entirely upon AMOS's own special talents – scrolling, blitter objects, music, AMAL etc – and to show you how these talents can be applied to games and demo programming. Rest assured that all the commands that are AMOS-specific will be documented within this book, so if you do already have even a basic (pardon the pun) understanding of the BASIC language (even if you've never programming in AMOS before), then this book is all you will need.

---

## Planning your program

Always plan your program before committing yourself to code. When I was first taught to program, it was constantly drummed into us that you should never approach a large programming project by sitting down in front of a computer terminal. As my old programming lecturer would say 'coding is a fairly minor aspect of the programmer's art'. So what do you do for the rest of the time? Well, the first thing you do is to turn off your

Amiga, pick up a pen and paper, go and sit in a comfy chair and plan the whole project out on paper!

Planning a program isn't as long-winded as it sounds. You don't, for example, write all your code out on paper (this would be defeating the object of not coding directly!). At this stage in development, all you need to do is to think about what will be involved. Let's take the example of a game (PacMan, say). If you were to simply sit in front of your Amiga and start coding, you'd soon run into problems. But if you sit and plan it first, you'll find that games (and indeed most programs) aren't that complex at all.

### 'Pseudo code'

You should start by breaking the game down into a series of steps, each of which describes a single operation that must be performed. In the case of our PacMan game, we'd end up with a list of steps like the following.

#### Start of game

```

Move PacMan according to joystick position
Increase score if PacMan eats 'dot' or 'Power Pill'
Make PacMan invincible if he eats 'Power Pill'
If PacMan is invincible
    Move Ghosts away from PacMan
Else
    Move Ghosts towards PacMan
End If
Check that PacMan and Ghosts haven't collided
If they have collided
    If PacMan is invincible
        Kill Ghost
    Else
        Kill PacMan
    End If
End If
End of game loop

```



On first impressions, our list looks very similar to conventional program code and, to be honest, it's supposed to. This is what programmers call



‘pseudo code’. Pseudo code won’t actually work if you type it into AMOS, though (try it if you don’t believe me!) – it’s simply designed as a guide to the programmer to show them how the program is structured in plain English. Pseudo code is very much simplified. It doesn’t, for example, tell the programmer how to actually write a routine that moves PacMan according to the player’s joystick position.

Once you’ve got your game (or program) broken down into these simple steps, you can then break each step down further. Here’s the pseudo code that moves PacMan according to the player’s joystick position.

Start of routine

Check player’s joystick

If the joystick is pushed right

    If PacMan can move in this direction

        Move PacMan

    End If

End If

If the joystick is pushed left

    If PacMan can move in this direction

        Move PacMan

    End If

End If

If the joystick is pushed up

    If PacMan can move in this direction

        Move PacMan

    End If

End If

If the joystick is pushed down

    If PacMan can move in this direction

        Move PacMan

    End If

End If

End of routine



Once again, the pseudo code looks very similar to conventional program code, but it still isn’t program code. But now we’ve increased the amount of detail that it shows, you can start to see how your AMOS code would

be structured. Indeed, if you're a pretty experienced coder, you could probably translate this pseudo code into working program code.

## **Subroutines & modular programming**

Without even realising it, we've also hit upon another programming concept that is very, very important – subroutines. A subroutine is essentially a mini-program that is embedded in a larger program. It is designed to perform a single operation (in this case, moving PacMan). A typical AMOS program should contain many of these subroutines, each of which is responsible for handling a particular aspect of your game or program. Writing a program using subroutines is what is known as 'modular programming' as opposed to the 'linear' programming techniques that most amateur programmers use.

Modular programming allows you to write a program in such a way that new modules (subroutines) can be added and old modules removed without having to rewrite large sections of code to cope with the changes. By splitting your program up into subroutines, you can also quickly and easily see how the program flows, making your code much more readable in case you need to make changes at a later date.

## **Is your code 'readable'?**

Here's another very important concept – code readability. OK, so it may only be you that ever gets to see your source code (source code is just a programmer's term for program code), but it's very important to structure your code in such a way that it will remain completely readable no matter how long you leave it.

If you were to write for a commercial software house (which is the ultimate aim of most amateur programmers), they would insist that your code remains readable so that in the event that you were to suddenly leave (for an unplanned lifetime holiday or whatever), they could pass your code onto another programmer who could continue working on it. If you write your code in a totally unreadable way, the new programmer would probably be forced to dump your code and start from scratch!

Take Electronic Arts' Deluxe Paint, for example – although DPaint was originally written by Dan Silva (a very nice chap who I was lucky

enough to meet!), later versions of DPaint were passed onto Lee Taran (who, contrary to popular belief, is actually a woman). If Dan had not made his code readable, Lee would have had a real hard time of it!

## Code comments

AMOS provides the programmer with a number of useful facilities that can make the task of making code readable that bit easier. The first of these is the 'REM' command that simply inserts comments (REMARKS) into program code. Make use of these as much as possible. For example, try to use REM commands at the start not just of your program, but each and every subroutine too. Something like this is always a good idea.

```
REM *** MoveSprite Routine
REM *** This routine controls the movement of the
REM *** player's sprite.
```



Comments come in handy when you start to write subroutines that need to be passed parameters (values). If, for example, you had a routine that needed to be passed three parameters, you could put a comment at the start of the routine about what the three variables are for. For example:

```
REM *** MoveSprite Routine
REM *** Expects to be fed three variables - Name$, X and Y
REM *** Name$ = Name of player
REM *** X      = X coordinate of player's sprite
REM *** Y      = Y coordinate of player's sprite
```



Putting comments at the start of a routine helps to make the overall flow of a program that bit easier to understand – but it doesn't make the workings of the code any easier. It might, therefore, be worth inserting code comments 'on the fly' just to increase the readability of your code.

## Code indentation

Another very useful programming technique that is actually built into AMOS is code indentation. Code indentation is basically a technique used to make code easier to understand by formatting the layout of the program. By simply indenting code within a particular control structure using the tab key, you'll be amazed just how much more readable AMOS

code becomes. To demonstrate this, let's take a look at a short AMOS program (and yes, it will work if you type it into AMOS).

```
Input "Please enter your age "; AGE
If AGE < 16
Print "You're too young to smoke!"
Else
If AGE > 17
Print "You're old enough to watch an '18' film!"
Else
Print "But you're still not over 18!"
EndIf
If AGE > 59
Print "Isn't it time you retired?"
EndIf
EndIf
```



OK, so this program is still straightforward enough to understand. But we can make it much more readable by indenting sections of code, either manually (using the 'TAB' key) or let AMOS do it for you by selecting 'Indent' from the menus? The result is a much more 'readable program':

```
Input "Please enter your age "; AGE
If AGE < 16
    Print "You're too young to smoke!"
Else
    If AGE > 17
        Print "You're old enough to watch an '18' film!"
    Else
        Print "But you're still not over 18!"
    EndIf
    If AGE > 59
        Print "Isn't it time you retired?"
    EndIf
EndIf
```



## AMOS procedures

One of the most useful facilities that AMOS Basic offers the programmer for making code readable is the good old ‘procedure’. Most BASIC dialects offer a procedures facility in one form or another, but AMOS’s are particularly powerful. A procedure isn’t like a normal command, though – on their own, procedures don’t actually do anything at all. What they can do, however, is to organise program code into a series of subroutines that can be called from your main program with a single command. Indeed, once a procedure has been defined, it can be called almost as if it were an AMOS command in its own right.

Defining a procedure is very simple indeed. All you need to do is to start the section of code that you wish to be a procedure with the following command (replace <procname> with your own procedure’s name).

---

```
Procedure <procname>
```



Once you’ve issued this command, you can then enter all the code that is to be contained within the procedure and then, once this is done, you need to mark the end of the procedure using the ‘End Proc’ command. Here’s a little demonstration program that shows how procedures work. Feel free to type it in and run it within AMOS if you so wish.

```
REM *** Procedures demo ***
```

```
MYPROC
```

```
End
```

```
Procedure MYPROC
```

```
    Print "Hello there!"
```

```
End Proc
```



## Handling variables

One of the great advantages of procedures is the fact that any variables that you define within a procedure are kept completely separate from both your main program and any other procedures you may have defined.

You can, therefore, use the same variable name within several procedures without having to worry about one procedure changing the value of a variable defined by another procedure. In this respect, any variables that you define within a procedure are what is known as ‘local’ variables. For a demonstration of this, try entering the following program.

```
REM *** Local Variables Demo ***
```



```
NAME$ = "Frank Smith"
```

```
MYPROC
```

```
Print NAME$
```

```
End
```

```
Procedure MYPROC
```

```
    NAME$="John Bloggs"
```

```
    Print NAME$
```

```
End Proc
```



**'Local'  
variables**

OK, so there's nothing special about this little demo, but it does demonstrate beautifully how the variables used within a procedure are completely separate from the main program. If this was not the case, the value of the variable 'NAME\$' that we defined at the start of the program would have been changed when the program called the procedure 'MYPROC'. But, because the procedure creates its own local variable under the same name, the value of the first variable is not changed.

One thing to note, however, is that local variables are only temporary. If you define a variable within a procedure and then return to the main program, you can't call the procedure again and expect it to have remembered the value of a local variable used within that procedure. If you do need to retain the value of a variable defined within a procedure, you should either return the value to the main program (using 'End Proc[<variablename>]' and the 'Param' command) or you should define a 'global' variable.

Global variables come in particularly handy when you wish to define a variable that can be accessed by any part of your program code. Say, for



example, you wanted every procedure within your program to automatically know the whereabouts of a sprite on screen. This is done using the ‘Global’ command. You should be aware, however, that a variable can only be defined as global within the main part of your program. Also, the global command does not actually create the variable for you – this must be done before you issue the command within your program. Let’s take another look at the demo we used above, but this time let’s use define the ‘NAME\$’ variable as a global variable.

```
REM *** Global Variables Demo ***
```



```
NAME$ = "Frank Smith"
Global NAME$
```

```
MYPROC
Print NAME$
End
```

```
Procedure MYPROC
    NAME$="John Bloggs"
    Print NAME$
EndProc
```

When you run this version of the program, instead of getting ‘Frank Smith’ and ‘John Bloggs’ printed on the screen, you should get ‘John Bloggs’ printed twice. By defining the ‘NAME\$’ variable as global, instead of creating a local variable that is separate from the main program, the procedure ‘MYPROC’ actually changes the value of the ‘NAME\$’ variable defined within the main program.

If you do need to keep certain variables local but you’d still like a particular procedure to have access to those variables, then AMOS allows us to give a procedure access to those variables using the ‘Shared’ command. This command must live within the procedure that needs access to the variables and the variables must have been defined beforehand within the main program. If you simply declare a set of variables as ‘Shared’ within the procedure without defining them first, AMOS will simply treat them as local variables. Here’s another demo:

```
REM *** Shared variables demo ***
```

```
NAME$ = "Frank Smith"
```

```
MYPROC
```

```
End
```

```
Procedure MYPROC
```

```
    Shared NAME$
```

```
    Print NAME$
```

```
End Proc
```



### Parameters

AMOS also allows you to define a set of parameters that can be passed to a procedure. When you come to call a procedure that expects parameters, it's a bit like passing parameters to a normal command but, instead of simply entering the parameters after the command name within your main program, they need to be enclosed within a set of square brackets ('[' and ']' symbols). The great thing about this approach is that it allows you to share the value of any given variables with a procedure without having to worry about whether the procedure will actually change them. In effect, passing parameters to a procedure simply provides it with working copies of variables that are passed. Here's yet another demo:

```
REM *** Parameters demo ***
```

```
NAME$ = "Frank"
```

```
MYPROC [NAME$]
```

```
Print NAME$
```

```
End
```

```
Procedure MYPROC [NEWNAME$]
```

```
    NEWNAME$=NEWNAME$+" Smith"
```

```
    Print NEWNAME$
```

```
End Proc
```



One thing to note from this short demo is that the name given to the parameter can be completely different from that passed to the procedure. AMOS allows us to do this because the variable defined by the procedure to hold the parameter that we passed is a local variable that is simply



used to temporarily hold the value that is passed by the main program. Once the procedure has finished and control has been passed back to the main program, this local variable is then forgotten along with any changes that may have been made to it.

# Screens

- Screen modes (including AGA)
- Opening screens
- Screen management
- Screen palettes
- Resizing and positioning screens
- Loading and saving screens

**S**o far within this book we haven't really covered anything that can't already be found within other BASIC programming languages, both on the Amiga and indeed lesser machines. Procedures, integrated programming environments and even 3D and compiler extensions can also be found running on other BASIC dialects. But – as we all know – AMOS is more than a bit special. Then again, you shouldn't need me to tell you that – after all, you've already bought AMOS (and this book!).

One of the most fundamental aspects of AMOS programming that you must be aware of is that of screens, those wonderfully colourful thingies (who needs jargon when you've got such a great grasp of the English language) you see on your Amiga's monitor or TV. Screens are very important within AMOS – without a screen, your program won't be able to display anything. A screen acts as a portal between the computer and the user and lets it communicate with us humans.

As an Amiga user, you probably already know just how powerful the Amiga's screen hardware really is – not only can you create screens in a number of different resolutions and with a maximum of 4096 colours (or 262,000 on an AGA machine), but you can even mix and match screens so that the display you see on your monitor or TV comprises several (completely independent) different screen zones, each with its own resolution and colour palette. What's more, because AMOS bypasses the Amiga's operating system, all screen handling is done at hardware level, therefore ensuring that everything runs as fast as possible.



#### **Amiga screen hardware**

You've probably already experienced this hardware phenomenon if you've played around with the Amiga's Workbench – if you load a program such as DPaint, you can literally drag the program's screen down to reveal the Workbench screen behind it. If you're not aware of the technicalities involved, though, you may have taken this for granted. What the Amiga is actually doing is displaying two entirely separate screens within the same display.

The Amiga isn't restricted to just a couple of screens, though – AMOS, for example, can 'open' and display up to eight of these screens, which can then be moved about, resized and even scrolled to your heart's content.

The Amiga's video hardware allows programmers to split the screen into a number of independent sections, each with their own resolutions and colour palettes. A good example of this is the Amiga's Workbench which can be dragged down to show a screen running behind it.



The key to all this screen jiggery-pokery is a little sliver of silicon built into the Amiga's Denise chip (renamed 'Lisa' on AGA machines) called the 'copper' which has nothing whatsoever to do with the great British Bobby but a great deal to do with screen synchronisation and display. We're not trying to put the Commodore Hardware Reference Manual to shame here, so I won't say too much about the technicalities involved. Suffice to say that all the hard work is handled by AMOS, so you can take advantage of all this powerful screen hardware without ever getting your hands dirty.

It's important to know a little about the sort of screen combinations that the Amiga's hardware is capable of. We won't be covering them all here though – only those that are addressable by AMOS. With the advent of the Enhanced Chip Set (ECS) and the new super swanky Advanced Graphics Architecture (AGA) chips sets, the Amiga is now capable of handling far more screen modes, but AMOS can only take advantage of those present in the original chip set built into the A500 and 2000 series Amigas. By the time you read this, Europress may have launched the promised AGA-compatible version of AMOS Professional, which will be able to handle the AGA chip set's new VGA-style 256-colour and 262,000-colour HAM8 screen modes as well as the SuperHiRes modes originally found in the ECS chip set upgrade.



**AMOS and AGA**

---

## Screen combinations

The standard Amiga chip set is capable of basically four different screen modes, all of which have their own pros and cons.

### 1. Low resolution

The most basic of these is low resolution mode, the screen mode used by 95% of all Amiga games. Low resolution offers 320 pixels across by 256 (or 200 on an NTSC Amiga) pixels down. Low resolution mode is capable of displaying a maximum of 32 colours, although extra colours can be squeezed out of the machine using Extra HalfBrite and the rather quirky HAM modes.

### 2. Extra Half Brite

Extra Half Brite (yes, 'Brite' is spelt correctly!) is a rather strange screen mode that doubles the maximum number of colours available from 32 to 64 colours. Unfortunately, the extra 32 colours are not independent – instead, they are simply copies of the first 32 colours, but the Amiga's hardware effectively halves their brightness (hence the name, Extra Half Brite). HAM mode is a rather useless mode that is sometimes useful for displaying digitised pictures but little else. It can display the entire 4096 colour palette of non-AGA machines (don't forget that AGA offers a 16.7 million colour palette!) on-screen at once. This mode is very complex, so we won't be covering it in any great detail.

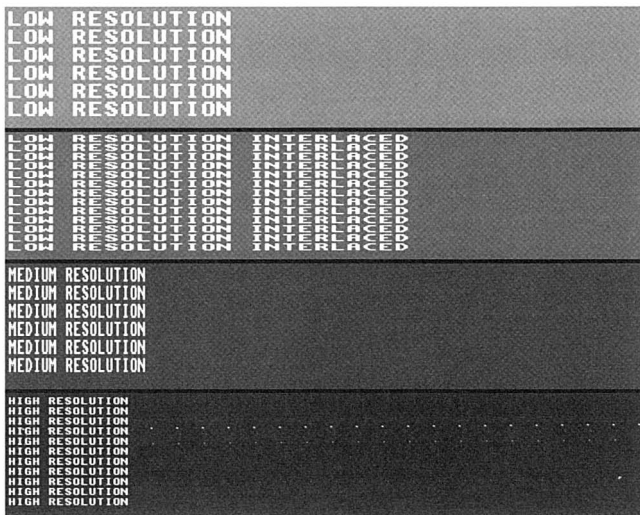
### 3. Medium resolution

Next up comes medium resolution, which is the default screen mode used by the Amiga's Workbench. Medium resolution offers 640 pixels across by 256 pixels down. Unless you're lucky enough to own a swish AGA machine, the maximum number of colours that the Amiga can handle is 16. Once again, until Europress launch an AGA upgrade for AMOS, this is the maximum number of colours that we can use regardless of the type of chip set you have inside your Amiga.

### 4. Interlace mode

Finally, we have interlace mode, which effectively doubles the vertical resolutions of both the low and medium resolution screen modes. Low resolution therefore increases to 320 by 512 (that's 2 x 256) and medium

Until the AGA version of AMOS Professional is released, AMOS supports four basic screen modes – low, interlaced low, medium and high resolution.



resolution mode increases to 640 by 512 pixels. Although we refer to an interlaced low resolution screen as ‘low resolution laced’, medium resolution laced screens are called ‘high resolution’. Confusing, I know, but bear this in mind. Interlacing a screen mode doesn’t have any effect on the number of colours that it can display – a 16-colour medium resolution screen that is reopened as a high resolution screen can still only display a maximum number of 16 colours.

You may have noticed that earlier we mentioned ‘NTSC’ mode. This has absolutely nothing to do with the actual screen modes itself, but refers instead to the vertical resolution of the screen. NTSC is a television standard that is used in the States and is slightly different to the PAL system we have over here. OK, I know that the Amiga’s not a television, but it does have video hardware that has to work in conjunction with the NTSC or PAL monitors and TVs used over here and across the pond. Put simply, NTSC Amigas can only display a maximum of 200 vertical lines in non-interlaced mode and 400 lines in interlaced mode. PAL Amigas, on the other hand, have a higher resolution — 256 pixels in non-interlaced modes and 512 pixels when interlacing is used.



So why’s this so important? Well, if you intend to write software that will be distributed to other users, you need to be aware that whilst your PAL

software will work perfectly well in Britain, American users won't be able to see all of the screen. If you intend writing commercial software, most software houses will insist that your games run under NTSC resolutions.

Sounds complicated? You ain't seen nothing yet! Finally, we have overscan, which is a special feature offered by the Amiga's screen hardware that allows you to remove the border around all standard screen modes. Overscan is primarily designed for desktop video, but it can also be very effective when used within games. By removing the border around the screen, the Amiga allows the screen to fill the entire display rather like the coin-op machines you'll see down your local arcades. Anyway, here's a quick table for your reference:



**Overscan**

<b>Screen Format</b>	<b>Resolution *</b>	<b>OverScan</b>	<b>Max Colours</b>
PAL Low Resolution	320 x 256	368 x 283	32 **
PAL Low Resolution Laced	320 x 512	368 x 566	32 **
PAL Medium Resolution	640 x 256	736 x 283	16
PAL High Resolution	640 x 512	736 x 566	16
NTSC Low Resolution	320 x 200	368 x 241	32 **
NTSC Low Resolution Laced	320 x 400	368 x 482	32 **
NTSC Medium Resolution	640 x 200	736 x 241	16
NTSC High Resolution	640 x 400	736 x 482	16

\* Resolutions are expressed as width x height

\*\* These three screen modes can use extra colours using Extra Half Brite and HAM modes.

---

## **AGA screen modes**

As I said earlier, Europress hadn't launched the AGA version of AMOS Professional at the time of writing, but I will give you a quick overview of the new improved screen modes that the AGA chip set has to offer.

### **Extended Palette**

The new AGA chip set dramatically increases the colour palette offered by the Amiga from 4096 colours to a massive 16.7 million. This effectively means that for every 1 colour offered by the old chip set, AGA can produce an extra 4096!

### **New Screen Modes**

The only real new screen mode offered by AGA is SuperHiRes, which was actually built into the Enhanced Chip Set used on the A500 Plus and A600. SuperHiRes mode offers a maximum resolution of 1280 pixels across and either 256 (non interlaced) or 512 (interlaced) pixels down. SuperHiRes isn't very practical for games, though – even on an A4000/040, SuperHiRes mode is rather slow.

### **VGA Compatibility**

Under AGA, the maximum number of 'real' (i.e. non-HAM) colours that can be displayed on any screen has been increased to 256 colours. This really is genuinely useful to games programmers, so the new AGA-compatible AMOS Professional will definitely support this facility.

### **HAM-8**

Another new screen mode offered by AGA is HAM-8, which is a vastly extended version of the original chip set's HAM mode that can display a maximum of 262,144 colours on-screen at once on any screen (yes, even a SuperHiRes screen!). It's very impressive, but oh-so-slow. Ideal for flash title screens, but totally useless for game screens.

### **Deinterlacer**

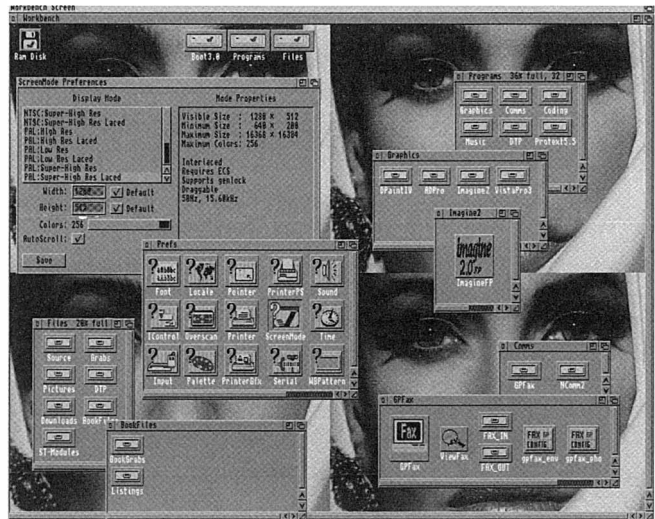
All AGA machines have what is called a 'deinterlacer' that removes the annoying flicker that is an unavoidable side effect of interlaced screens. It works by doubling the horizontal scan rate of any screen from 15 KHz to 29 KHz. As all TVs and RGB monitors can only display images with a



When the AGA version of AMOS Professional is finally released, you'll be able to open screens with up to 262,000 colours with a maximum resolution of 1280 by 512 pixels!



**AMOS Pro AGA**



horizontal scan rate of 15 KHz, you'll need a VGA or multisync monitor to take advantage of this. DeInterlaced screens are referred to as 'DBL' modes (short for 'Double'). Once again, DBL modes are pretty useless for games as most users don't even have VGA monitors!

## Opening screens

Phew! Thank heavens we've got all that theory out of the way. By now you should have a pretty sound understanding of the sort of screen combinations available on the Amiga and the sort of screens that AMOS itself can handle. Now that all that theory has been firmly implanted into your brain, we can actually move onto some AMOS coding.

In order to display anything on your Amiga's TV or monitor, you need to open a screen using the AMOS command 'Screen Open'. By default, AMOS already kindly opens up a low resolution NTSC-style screen with 16 colours, but we can open our own in any of the screen modes shown within the table above. What's more, a maximum of eight such screens can be opened simultaneously with their own resolutions and colour palettes. AMOS keeps track of all these screens using a screen number between 0 and 7 – the default screen, for example, has a screen number of 0. This screen number is very important once you start to create

displays that use more than one screen simultaneously, as you'll need to tell AMOS which screen to use by passing it the screen's number.

The Screen Open command needs to be passed a number of parameters in order to be able to successfully open a screen. Here's the command in all its glory plus a brief description of what each parameter does:

---

Screen Open Screen Number, Width, Height, Colours, Mode



**Screen Number** The screen number is an integer between 0 and 7 which is used by AMOS to distinguish multiple screens. The default screen opened by AMOS has a screen number of 0. If you therefore attempt to open a screen using this number, AMOS will close its default screen and open a new screen in its place using the parameters that you define.

**Width** This is an integer value that defines the width of your new screen in pixels. For a detailed breakdown of the sort of width combinations available, refer to the table earlier in this chapter. If you pass a value greater than 320 (for low resolution screens) or 640 (high resolution), AMOS will automatically turn on horizontal overscanning. You're not just restricted to those values detailed within the table above, however. Thanks to AMOS's powerful screen scrolling hardware (which we shall be covering later), it's perfectly possible to create screens that are much larger than can be displayed on your TV or monitor. When you start using hardware scrolling, this feature will become very important indeed.

**Height** The height parameter defines the vertical resolution of your screen in pixels. Once again, you're not restricted to the standard 256 or 512 (interlaced) screen height settings. If you define a height greater than these values, AMOS will automatically turn on vertical overscanning.

**Colours** The colours parameter tells AMOS how many colours you'd like your new screen to use. Normal non-AGA rules apply here, so there's no point trying to use more than 16 colours in medium or high resolution, for example. If you intend opening a low resolution (both interlaced and non-interlaced) screen mode, however, you can also pass a value of 64 (or Extra Half Brite) or 4096 (for HAM).



### Colour and memory

It's important to remember too that the more colours you allocate to a screen, the more memory it uses. A standard PAL low resolution screen with just 2 colours will only eat up only 10K. If you use 32 colours, however, this increases to 50K! What's more, the more colours you use, the slower AMOS's drawing and blitter object (Bob) commands become. Unless you're using an AGA machine (these can happily handle just about any non-HAM screen mode with little or no speed decrease), try to restrict your games to just 16 colours. OK, so they won't be as colourful, but at least they'll remain fast.

**Mode** The mode parameter is used to tell AMOS what type of screen you wish to open. There are three alternatives – Lowres, Hires, Laced. Lowres creates a non-interlaced low resolution display (320 pixels across) and Hires creates a non-interlaced medium res display (640 pixels across).

The Laced option must be used in conjunction with the Lowres and Hires parameters to double the vertical resolution of these two screen modes using interlacing. This option is not available in very early releases of AMOS (if your version of AMOS can't handle laced screens, then get your hands on an AMOS updater disk!). To create a low res interlaced display, you'd therefore pass 'Lowres+Laced' for the mode parameter.

As you can see, the Screen Open command is certainly straightforward enough. If, for example, you wanted to open a low resolution PAL screen with 32 colours as screen 0, you would use the following line.

```
Screen Open 0,320,256,32,Lowres
```



## Screen management

Once you start opening more than one screen simultaneously, it's all too easy to lose track of which screen AMOS is currently working with. AMOS can only ever work on a single screen at any one time, regardless of the number of screens that you've opened. By default, AMOS also treats the last screen that you opened as the 'current screen' unless you tell it otherwise. Most of the drawing commands that AMOS offers do

not allow you to specify which screen that are to operate on, so it's down to you to make sure that AMOS knows which screen you wish to work with before attempting to carry out any form of drawing or blitter operations.

AMOS allows you to change the current screen using a very simple command called the 'Screen' command. All you have to do is to tell the command which screen (using the screen number) you wish AMOS to treat as the current screen and it goes away and does the rest. Say, for example, you had opened three screens number 0, 1 and 2, but (because screen 2 was the last to be opened) AMOS was performing all drawing and blitter operations into screen 2, despite the fact that you actually wanted to draw into screen 1. You would therefore use the line 'Screen 1' to instruct AMOS to switch over to screen 1.

Opening and switching between multiple screens is all very well and good, but you also get into the habit of closing them once they are no longer needed. Closing a screen removes it from view and returns the memory it used to the system so that it can be used for other things (opening other screens, for example). This is very important if you intend to write software that can be run on Amiga systems with less than 1 Mb (do such machines still exist?) – if you start opening screens willy nilly, you may find that your program will crash on Amigas that aren't fitted with RAM expansions, due to a lack of valuable memory. Even if you are lucky enough to own an expanded Amiga, you'll be able to maximise the number of potential users for your software if you make it 'memory friendly'. Not only that, but – as my old mum used to say – you should always clean up after yourself!



### **Multiple screens**

Closing a screen is even simpler than opening a screen. Unlike the Screen Open command, all you need to do is to pass the number of the screen that you wish to close to the Screen Close command – if you wanted to close screen 2, you would use the line 'Screen Close 2'. The following listing demonstrates all three of the screen commands covered so far:

```
Rem *** Screen handling demonstration
Rem *** Filename - Screens.AMOS
```



```
Screen Open 0,320,100,32,Lowres
Screen Open 1,640,200,4,Hires+Laced
Screen Open 2,640,16,2,Hires
```

```
Screen Display 1,,147,,
Screen Display 2,,250,,
```

```
Screen 2
```

```
Locate 0,0 : Centre "Press <SPACE BAR> to swap screens!"
Locate 0,1 : Centre "Press 'Q' to Quit"
```

```
SCR=0 : Screen SCR
```

```
Repeat
```

```
  KEE$=Inkey$
```

```
  If KEE$=" "
```

```
    SCR=SCR+1
```

```
    If SCR=2
```

```
      SCR=0
```

```
    End If
```

```
    Screen SCR
```

```
  End If
```

```
  Print "AMOS! ";
```

```
Until Upper$(KEE$)="Q"
```

```
Screen Close 0
```

```
Screen Close 1
```

```
Screen Close 2
```

---

## Screen palettes

When a screen is first opened, AMOS automatically allocates a preset selection of colour settings to the screen's palette. These can be changed quite easily to suit your own particular needs using either one of the two palette editing commands AMOS has to offer. These are 'Palette' (for setting up colour palettes 'en masse') and the imaginatively named



### SCREEN command

The 'Screen' command can be used to instruct AMOS which screen you wish all drawing operations to be carried out on.



'Colour' (for altering single colours). Until the AGA release of AMOS Professional finally becomes available, you can choose from any of the 4096 colours offered by the standard and Enhanced chip sets. The AGA version promises to extend this so that your screens can draw upon the massive 16.7 million colour palette offered by the AGA chip set.



### Hex values

Both commands need to be fed hex values that define colours as three-digit hex numbers. Don't worry too much about having to work with hex – yes, I know that it's usually very complicated, but you don't need a degree in computer science to work out the hex values required by AMOS! As you will probably already know, all colours are specified in terms of their red, green and blue content. Each of these three 'colour components' can be set to one of 16 different 'intensities' ranging from 0 (black) to 15 (maximum intensity). Solid white, for example, is all three colour components set to maximum – a value of 15 for red, 15 for green, 15 for blue – whereas black is all three colour components set to zero.

For AMOS to understand a colour setting, these three colour component values must be converted to hex format and then combined into a three-digit hex value. As a value of 15 is represented using the hex value \$F, solid white would be hex \$FFF (15,15,15). Note the order that the hex values are combined – red first, followed by green and then blue. If you had a colour with a red value of hex \$6 (decimal 6), a green value of hex \$F (decimal 15) and a blue value of hex \$A (decimal 10), the colour would be expressed as \$6FA.



**Converting  
decimal-hex  
values**

The AGA chip set, however, extends the palette settings so that each colour component can be set to a value between 0 and 255, allowing the full 16.7 million colour palette to be accessed. In hex terms, this basically means that instead of expressing a colour as three hexadecimal digits, an AGA colour is defined using six digits — two for the red component, two for green and two for blue. An AGA palette setting of red 255, green 140 and blue 20 would therefore be written in hex as \$FF8C14. If you're unsure about converting decimal values to hex, you can make life very easy by using the AMOS 'Hex\$()' function. For example, entering 'Print Hex\$(140)' would cause AMOS to print the value '\$8C' on the screen.

### **'Palette' command**

Anyway, back to the 'Palette' and 'Colour' commands. First, the Palette command. As its name suggests, this command allows you to define a screen's colour palette. Unlike the Colour command we'll be covering next, the palette command allows you to set up a whole string of colour registers in one go. The format of the Palette command is simple enough — all you do is to enter the command followed by a string of three-digit hex numbers that define the colour palette of the screen. All the colour settings must be entered in order, though, so it's no good trying to set up colour register 4 before colour register 3. Similarly, each colour setting must be in the exact position so that AMOS knows which colour register the setting must be loaded into. For example, if you wanted to change just colour registers 4 and 5, you would enter 'Palette ,,\$FFF,\$FA6' — note the commas that effectively blank off the first three colour registers.

### **'Colour' command**

Finally, we have the 'Colour' command, which is very similar to the Palette command, but works on only one colour register at any one time. What it does offer, however, is the ability to set any colour register 'on the fly' without having to suffer the hassle of inserting commas to mask off any colour registers that precede the one you wish to change. The format of the Colour command is 'Colour Register, Hex Value'. If you wanted to change colour register 17 to hex \$6A5, for example, you would enter 'Colour 17,\$6A5'. Here's a quick demo program that shows both techniques in action:

```
Rem *** Palette and Colour Demonstration
Rem *** Filename - ScreenPalette.AMOS

Screen Open 0,320,100,8,Lowres
Flash Off

Screen Open 1,320,100,8,Lowres
Flash Off
Screen Display 1,,155,,

Screen 0
Palette $666,$FFF,$6AC,$222,$F00,$F0,$F,$FF
_SHOWCOLOURS[8]

Screen 1
Colour 0,$B2C : Colour 1,$FF0 : Colour 2,$CA6
Colour 3,$2B4 : Colour 4,$F0F : Colour 5,$F0
Colour 6,$BD2 : Colour 7,$28F
_SHOWCOLOURS[8]

End

Procedure _SHOWCOLOURS[NUMCOLS]
  Cls 0 : Pen 1 : Paper 0
  Print "Here's your colour palette"
  Print
  For C=0 To NUMCOLS-1
    Paper 0 : Print "Colour";C;" = ";
    Paper C : Print "      "
  Next C
End Proc
```



---

## Resizing and positioning screens

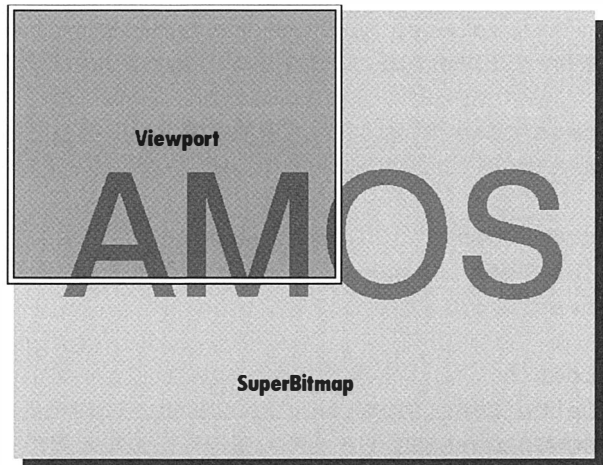
Once a screen has been opened, it can be positioned and even resized on your monitor or TV screen using the AMOS command 'Screen Display'. The screen display command comes in particularly useful when you start to open more than one screen simultaneously — because all new screens





### SCREEN DISPLAY command

The Screen Display command allows you to alter the position and size of a screen's viewport. A viewport acts as a sort of 'window' onto a screen's bitmap, allowing you to see sections of a bitmap that is much larger than the maximum resolution of the display hardware. When used with AMOS' hardware scrolling facilities, the viewport can even move around the bitmap.



### Screen arrangement

are automatically opened using the top left hand corner of the screen as the origin, they will overlap. You therefore need to use the Screen Display command to arrange them correctly on your monitor display. Due to a limitation in the Amiga's screen hardware, however, screens can only be arranged vertically. You cannot, therefore, have two screens side-by-side. If you attempt to put two screens next to each other, the screen that was opened last will completely cover the screen beneath it.

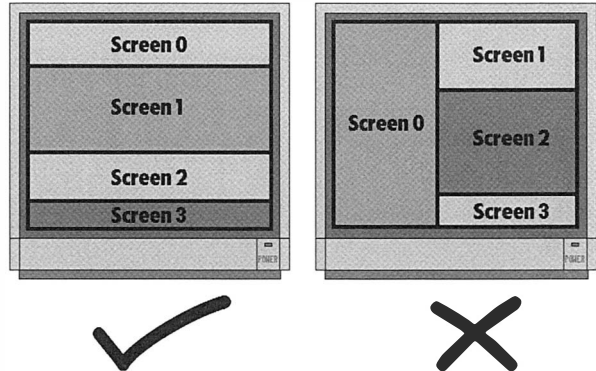


### Viewport

The Screen Display command can also be used to resize a screen. This isn't quite as straightforward as it may first seem, however. Whenever you resize a screen, it's not the actual screen you are resizing, but the 'viewport' that is used to display it. A viewport is best thought of as a sort of window that is used to view the contents of a screen. If you like, think of it as the window in a house – if you're inside the house, you can look through the window to see blitter objects and sprites outside in your garden (time to get out the greenfly spray, methinks!). The garden itself is much bigger than the window, so you can only see a tiny proportion of the garden at any one time. You can, however, alter the size of the window (using your trusty sledgehammer!) which will reveal more of your garden.

One thing worth noting, however, is that even if you do resize a screen so that it is effectively bigger than it was when it was first opened, the screen itself remains the same size. What you must remember is that

Several screens can be arranged vertically, but they cannot be arranged horizontally due to a limitation in the Amiga's hardware design..



### Screen size

when a screen is first opened, a section of memory is set aside to hold the screen's bitmap (the part of the screen that AMOS draws into). If the Screen Display command were to increase the size of the screen, it would also have to increase the size of the bitmap. The width and height parameters (the height parameter is discussed below) are really only designed to allow you to alter the size of the screen's 'viewport' so that screen bitmaps that are larger than the default 320/640 by 256/512 pixel screen sizes can be displayed in overscan format.

The Screen Display command really comes into its own when used in conjunction with AMOS's hardware scrolling command, Screen Offset (more on this in the next chapter). Because the Screen Offset command works by scrolling a bitmap that is larger than the physical display, the Screen Display command needs to be used to create a sort of 'window' that restricts how much of this 'superbitmap' we can actually see.

Let's take a look at the Screen Display command and its format:

Screen Display Screen Number, X, Y, Width, Height



**Screen Number** Once again, the Screen Number parameter is simply a pointer to the screen that you wish to manipulate. If, for example, you had opened a screen using a value of 2 as its screen number, you would pass a value of 2 to this parameter so that the Screen Display command knows exactly which screen you are referring to.



### Parameter values

**X** The X parameter is an integer value between 0 and 448 that controls where your screen is to be positioned horizontally. In practice, however, this range of values don't quite perform how they should (a bug in the Amiga hardware?). If you can, try to stick to values between 112 and 432. If you use values any higher or lower than these, strange things tend to happen!

Also worth noting is that the Amiga automatically rounds the X parameter to the nearest multiple of 16, so even if you do pass a value of 18, AMOS will treat it as a value of 16. It's not until the value reaches 32 that a changes will become visible. A stupid limitation imposed by the Amiga's hardware, but one you can get around quite easily using hardware scrolling (more on this in the next chapter, though!).

**Y** Just like the X parameter, the Y parameter controls the vertical position of the screen. The range for this parameter is 0 to 312, although even on a PAL system, any values higher than 300 will usually make the screen disappear completely. Note that the Y parameter doesn't have to conform to the silly 16-pixel limitation imposed by movements in along the X axis, so you're free to specify any value.



### Contracting/ expanding the viewport

**Width** Now here's an interesting parameter that can be used to create some quite interesting results. Simply by passing a value that is greater or smaller than the width of the screen you specified when it was first opened, the Screen Display command can quite literally collapse or expand the horizontal length of your screen's viewport. The top left hand corner of the screen is taken as the origin, so the viewport will collapse or expand from the right hand side.

Once again though, the value you pass will be rounded down to a multiple of 16, so the same rules that applied for the X parameter apply here as well.

**Height** Just like the Width parameter, the vertical size of your screen's viewport can be increased or decreased by passing a value that is greater or smaller than the screen's actual vertical size. Like the Y parameter, the 16 pixel rule does not apply, so the screen's viewport can be resized with single-pixel resolution.

Right, that's enough of the theory – let's put what we've learned into practice. The demonstration listing below opens a low resolution NTSC screen which can be moved around on your TV or monitor using a joystick connected to port 2 on your Amiga. Don't worry about the joystick-handling routine for the moment – we'll be covering these later:

```
Rem *** Screen Display Demonstration
Rem *** Filename - ScreenDisplay.AMOS
```



```
Screen Open 0,320,200,32,Lowres
```

```
Flash Off : Curs Off
```

```
Locate 0,10 : Centre "Move me with the joystick!"
```

```
Locate 0,12 : Centre "Press FIRE to Quit"
```

```
X=128 : Y=48 : Rem *** Default values
```

```
Repeat
```

```
  If Joy(1)=1 Then Y=Y-16
```

```
  If Joy(1)=2 Then Y=Y+16
```

```
  If Joy(1)=4 Then X=X-16
```

```
  If Joy(1)=8 Then X=X+16
```

```
  If X>432 Then X=432
```

```
  If X<112 Then X=112
```

```
  If Y>300 Then Y=300
```

```
  If Y<0 Then Y=0
```

```
  Wait Vbl
```

```
  Screen Display 0,X,Y,,
```

```
Until Joy(1)=16
```

```
Screen Close 0
```

If that wasn't enough, here's another listing that demonstrates the Screen Display command's ability to resize a screen. The listing below opens up a screen of exactly the same dimensions as our first demo and

continuously collapses and expands it both horizontally and vertically.  
Have fun!

```
Rem *** Screen Resizing Demonstration
```

```
Rem *** Filename - ScreenResize.AMOS
```



```
Screen Open 0,320,200,32,Lowres
```

```
Locate 0,10 : Centre "Press <Ctrl> and <C> to Quit"
```

```
Do
```

```
  For A=320 To 48 Step -16
```

```
    Wait 5
```

```
    Wait Vbl
```

```
    Screen Display 0,,,A,
```

```
  Next A
```

```
  For A=48 To 320 Step 16
```

```
    Wait 5
```

```
    Wait Vbl
```

```
    Screen Display 0,,,A,
```

```
  Next A
```

```
  For A=200 To 2 Step -2
```

```
    Wait Vbl
```

```
    Screen Display 0,,,,A
```

```
  Next A
```

```
  For A=2 To 200 Step 2
```

```
    Wait Vbl
```

```
    Screen Display 0,,,,A
```

```
  Next A
```

```
Loop
```

```
Screen Close 0
```

## Loading and saving screens

Finally, we move onto the subject of AMOS's support for the IFF graphics standard. As those of you in the know will confirm, IFF is a very handy method of storing a variety of different types of data in a

**IFF format**

common file format. IFF (which, for the nosy amongst you stands for ‘Interchangeable File Format’) has been adopted by virtually every Amiga software developer so that any files that you produce can be directly loaded into another package without modification, providing the software you are trying to load the files into is designed to handle that type of data. There’s little point, therefore, trying to load a music score into a paint package – even if the package does support IFF, a IFF music score still won’t load into a paint program.

So what’s this got to do with AMOS? Well, AMOS just happens to be IFF-compatible, that’s what! What this basically means to the average AMOS programmer is that the contents of any screen that you create in AMOS can be saved out to disk as an IFF file, allowing them to be loaded into an IFF paint program. More exciting, however, is the ability to load IFF pictures into an AMOS screen. This comes in particularly handy when you need to design title screens and even backgrounds for your AMOS games – because an IFF picture can be loaded directly into AMOS, you can use a paint program such as Electronic Arts’ Deluxe Paint IV to draw all your game graphics. AMOS may be the ultimate programming language, but it’s a fact of life that you’ll still need a paint program sooner or later.

The two commands used to load and save screens are ‘Load IFF’ and ‘Save IFF’ respectively. Their format is as follows:

---

**Load IFF "Filename", Screen Number**



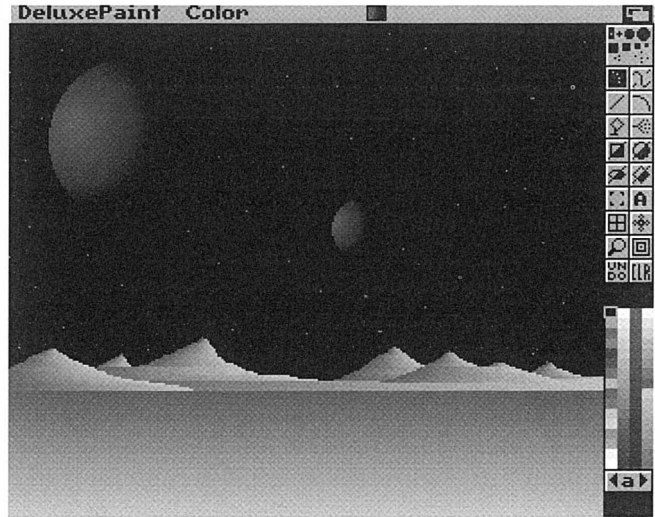

---

**Save IFF "Filename", Screen Number**



The Filename is simply a string that defines the name that the screen is to be loaded or saved under. Unless you’re loading or saving pictures from the current directory, you’ll also need to add in full path information – for example, to load a picture called ‘TitleScreen’ stored in a directory called ‘Pictures’ on DF0:, the filename string would be ‘DF0:Pictures/TitleScreen’.

AMOS's 'Load IFF' command allows you to design your game graphics within a traditional paint program such as *Deluxe Paint* and then pull them directly into your program.



### Screen types

The 'Screen Number' parameter is optional. If you don't use it, the current screen will be used, but it can often be handy to be able to tell AMOS exactly what screen you'd like saved when more than one screen is currently opened. In the case of the 'Load IFF' command, passing a screen number that points to a screen that is not currently open will cause AMOS to open one for you in the format required for the picture. If you attempt to load a picture into a screen of the wrong type (not enough colours, for example), you'll get an error message.

# Screen scrolling

- Screen synchronisation
- Hardware & software scrolling
- Superbitmaps and viewports
- Screen Copy scrolling
- Parallax scrolling
- Continuous scrolling
- Using screen 'blocks'
- AMOS TOME extension



If you've ever played shoot 'em up games such as Team 17's brilliantly executed 'Project X' or 'Alien Breed', then you've already witnessed a vivid example of screen scrolling in action. Scrolling may be a rather clichéd programming technique these days, but no-one could possibly doubt that it is a very powerful tool in the games programmer's arsenal. Screen scrolling allows you to move a screen along either the horizontal or vertical axis (or even both) to give the illusion of movement.

Take Project X, for example – the game starts with your ship flying through space but as you proceed through the game, your ship stays in the centre of the screen whilst the background graphics behind your ship move smoothly off to the left-hand side of the screen. It's a bit like the effect you'd see if you were looking out of the window of a moving car – although technically it's you that's doing the moving, the scenery on either side of the car appears to move past you. Screen scrolling works in a similar manner – in order to give the appearance of continuous movement without the player's sprite disappearing off the screen, the background is moved instead. The player's sprite is only allowed to move within the boundaries of the screen display. Even if the ship stays perfectly still, scrolling the background gives the illusion that the ship is actually moving.

Screen scrolling isn't just restricted to shoot 'em ups, however. It can be applied to just about any genre of game ranging from beat 'em ups and arcade adventures to platforms games – indeed, any game that needs to give the player the illusion of movement whilst keeping the player's sprite on the screen. Other examples of games that employ screen scrolling include Team 17's 'SuperFrog', Dino Dini's 'Goal' (or should that be 'Kick Off 4?'), 'Body Blows', 'Prince of Persia' etc.

Before we go any further though, it's worth mentioning the subject of screen synchronisation. Although we'll be covering this in great detail in the next chapter, I thought that now would be a good time to at least introduce a little bit of theory. All Amigas redraw their displays every 50th (for PAL systems) or every 60th of a second (for NTSC systems), depending upon the type of TV system used in your country. British Amigas are based around the PAL system, so the screen display is



**Screen syn-  
chronisation**

Screen scrolling is used to great effect within shoot 'em ups such as Team 17's excellent 'Project X'. By moving the background at speed past the player's sprite, you get the illusion of rapid movement without moving the player's sprite off the screen.



**WAIT VBL**  
command

redrawn 50 times a second. AMOS can scroll screens much faster than just 50 times a second, so you need to tie in your screen scroll with the TV standard using a command called 'Wait Vbl'. This command simply tells AMOS to halt every 50th of a second in order for the screen hardware to catch up with it. There's no point in getting your AMOS programs to scroll any faster than this – if you do, you'll get some very strange effects indeed. Check out the next chapter for more information on this important concept.

## Scroll types

As you'd expect, AMOS is fully equipped to handle the scrolling requirements of just about any game. Whether you want to scroll the entire screen or just a small section, AMOS is more than up to the job. What's more, because AMOS makes extensive use of the Amiga's blitter and built-in scroll hardware, you can scroll screens as fast (or as slowly) as you like.

AMOS offers basically two types of scrolling to the games programmer – software scrolling and hardware scrolling – each of which has its own particular pros and cons. Let's take a look at each in turn.

## Hardware Scrolling

Hardware scrolling gets its name from the fact that it uses the Amiga's own built-in screen scrolling hardware. Yes, even without a programming language as powerful as AMOS, the Amiga's hardware is capable of scrolling screens at unbelievable rates. Indeed, its hardware scrolling is so fast that it's actually possible to scroll a screen so fast that the Amiga's display hardware (the bit of circuitry that converts the screen image held in memory to an RGB or TV signal that can be displayed on a monitor/TV) cannot redraw the screen fast enough to show the scroll moving smoothly...

Hardware scrolling does have its limitations, though. Because it can only scroll whole screens, it does tend to be rather memory-intensive. Say, for example, you wanted to write a game like Team 17's Project X, which employed a scrolling background that was 20 low-resolution screens long. If you were to use hardware scrolling, the resulting bitmap would be over 200K in size – and that's only for a 2-colour bitmap! A 32-colour bitmap would be over a megabyte in size. Obviously, there are ways to get around this limitation, but it's down to you to code a more efficient hardware scrolling routine (or, better still, just type in the listing that you'll find later in this chapter!).



### Hardware scrolling limitations

## Software Scrolling

Software scrolling gets its name from the fact that it's not built into the hardware, but is actually a scrolling technique provided by AMOS itself. Instead of using the Amiga's own scroll hardware, AMOS's scroll commands make use of the Amiga's powerful blitter chip. The big advantage of this technique is that not only is it more memory-efficient, you're not restricted to scrolling whole screens – you could, for example, scroll just a tiny section of a screen's bitmap.

Unfortunately, software scrolling too isn't without its limitations. Although the blitter can operate much faster than the Amiga's own processor, it's still not as fast as the built-in scroll hardware. The blitter could theoretically scroll just as fast, but (as assembler programmers will tell you) setting up the blitter takes time. Although this done for you at machine code level, there's still a very slight delay between the time it takes to initiate the blitter and actually getting it to start doing its stuff.



### Software scrolling limitations

## Using hardware scrolling

Hardware scrolling is perhaps one of AMOS's most powerful facilities yet, considering its complexity, you'll be pleased to learn that incorporating hardware scrolling into your own AMOS creations is surprisingly simple. Indeed, all that is required to get things moving (if you'll pardon the pun) is a single command – Screen Offset.



### SCREEN OFFSET command

The Screen Offset command is very similar to the 'Screen Display' command that we discussed in the last chapter but, instead of moving the viewport, the screen bitmap that the viewport displays is moved instead by adjusting the position of the viewport's 'origin'. This origin tells AMOS where to start reading screen information. Let's take a look at viewports and screen bitmaps in a little more detail...

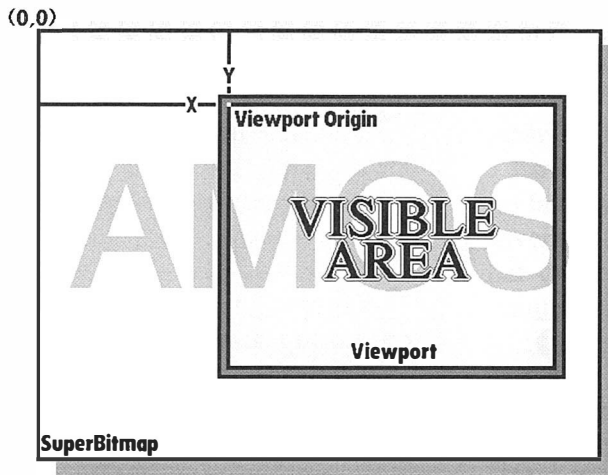
The Amiga's display is split into basically two components – the main screen display bitmap (the area that your programs draw into) and the display viewport. Imagine if you will that the screen bitmap is a landscape stretching for hundreds of miles and the viewport is a window in a house. If you were standing inside that house, the amount of scenery that you could see through the window would directly relate to the size and position of that window. Unless you had your eyes pressed against the glass, only a small proportion of the scenery outside could ever be seen. Imagine what it would be like if you could actually move that window around the wall, however. Because the window has been moved, you would be able to see previously invisible parts of the scenery outside. Dreamy stuff maybe, but this is exactly what the 'Screen Offset' command allows you to do.

All well and good, but there's one big problem. If the window is 320 by 256 pixels wide then the scroll area outside the window must be larger. After all, if the scroll area is scrolled one pixel to the left, a blank 1 pixel line will appear on the right. If the display then scrolled another pixel to the left, another blank line would appear. To get around this problem, you need to create what is known as a 'SuperBitmap'. A SuperBitmap is simply an image held in memory which is much larger than the rectangular area that you can see on your TV or monitor screen. Take a SuperBitmap that is 640 by 256 pixels in size, for example. If your

Hardware scrolling works by progressively changing the position of the screen's viewport so that a different section of a SuperBitmap is exposed.



'SuperBitmap'



viewport were 320 by 256 pixels, then only half of the SuperBitmap could be displayed at any one time. Using hardware scrolling, however, it is possible to change the viewport origin so that all of the SuperBitmap can be viewed. With a game that needs the background to be scrolled continuously in one direction, programmers cheat by looping the position of the viewport around the SuperBitmap so that when the scroll reaches the far end of the SuperBitmap, the viewport origin is reset to display the first part again. As we'll see later, this can be used to great effect.

Now that we've covered the theory, the time has come to put all that we've learned into practice. First though, let's take a look at the Screen Offset command in detail. The format of the command is as follows:

Screen Offset SCREEN NUMBER, X OFFSET, Y OFFSET



**Screen Number** The Screen Number parameter is a valid pointer to a currently open screen. If, for example, you had opened a screen as screen number zero, then a value of zero would be placed here.

**X Offset/Y Offset** These two values denote the X and Y position of the top left-hand corner of the viewport relative to the top left-hand corner of the SuperBitmap. If, for example, you had a 640 by 256 pixel

SuperBitmap and you wanted the viewport to display the middle 320 pixels (we'll ignore the Y parameter for now), the top left hand corner of the viewport would have to be placed at 160 pixels left of the start of the SuperBitmap. The middle 320 by 256 pixels would be displayed with 160 by 256 pixels hidden on both sides of the viewport. To get this kind of display, you would enter the line 'Screen Offset 0,160,0' (presuming the screen you wish to scroll is screen zero).

Anyway, here's a short of demonstration listing that shows the Screen Offset command in action:

```
Rem *** Screen Offset Demonstration
Rem *** Filename - ScreenOffset.AMOS

Screen Open 0,640,512,32,Lowres
Flash Off : Curs Off
Screen Display 0,128,40,320,256

For C=0 To 500
  X=Rnd(640)
  Y=Rnd(512)
  S=Rnd(50)+1
  Ink Rnd(32)
  Bar X,Y To X+S,Y+S
Next C

Ink 0,0 : Bar 22,80 To 300,100
Ink 2,0 : Text 30,93,"Move screen around with Joystick!"

SCRX=0 : SCRY=0

Repeat
  If Joy(1)=1 Then SCRY=SCRY-4
  If Joy(1)=2 Then SCRY=SCRY+4
  If Joy(1)=4 Then SCRX=SCRX-4
  If Joy(1)=8 Then SCRX=SCRX+4
```



```
If SCR<0 Then SCR=0
If SCRY<0 Then SCRY=0
If SCR>320 Then SCR=320
If SCRY>256 Then SCRY=256

Screen Offset 0,SCR,SCRY
Wait Vbl
Until Joy(1)=16

Screen Close 0
End
```

Scrolling a Superbitmap smoothly is simply a matter of adjusting the position of the viewport origin so that the SuperBitmap is shown progressively. In many ways, the demonstration listing above produces the same effect – the only difference being that instead of controlling the screen scroll through software, the joystick is used instead. It's worth noting too how the position of the screen viewport is continuously monitored so that it never goes above or below a set of minimum and maximum values.

The minimum value (0) is pretty obvious, but you may well be asking yourself why the above listing doesn't allow the viewport position to go above 320 pixels when the SuperBitmap is 640 pixels in size? This is because the viewport is positioned according to a viewport origin at the top left-hand corner of the screen, the position of viewport only dictates where the Amiga's screen hardware is to start reading screen data from. Our listing, for example, opens a viewport that is 320 pixels in size, so if you position the viewport at 320 pixels across, all the pixels up to 640 pixels across will be shown.



**Seamless  
continuous  
scrolling**

In order to produce smooth continuous scrolling in a particular direction, the Superbitmap must be made to wrap around. Obviously this will cause a noticeable jump if the second half of the screen doesn't exactly match the first, so most game scroll routines have two identical copies of the same image that link together to produce a seamless join. Here's another listing that demonstrates how to produce a continuous screen scroll similar to the type of scroll you might find in an arcade shoot 'em up.

```
Rem *** Hardware Scroll Demonstration
Rem *** By Jason Holborn
```



```
Screen Open 0,640,256,32,Lowres
Flash Off : Curs Off
Screen Display 0,128,40,320,256
```

```
Load Iff "AMOSBOOK:Pictures/HardScrollBackground.IFF"
Screen Copy 0,0,0,320,256 To 0,320,0
```

```
SCRX=0
```

```
Repeat
```

```
    SCRX=SCRX+4
```

```
    If SCRX=320 Then SCRX=0
```

```
    Screen Offset 0,SCRX,0
```

```
    Wait Vbl
```

```
Until Joy(1)=16
```

```
Screen Close 0
```

```
End
```

---

## Using software scrolling

There's no doubt that the Amiga's own hardware-based scrolling facility is the fastest form of scrolling AMOS has to offer, but it does have its limitations. As we have already seen, hardware scrolling is restricted to scrolling entire screens only. If you need to scroll only a small section of a screen, you need to use AMOS's own software scrolling commands.

Unlike hardware scrolling, which uses dedicated hardware to scroll the screen, software scrolling uses the Amiga's powerful blitter chip to shift large areas of the screen around at high speed. Many games programmers prefer software scrolling because it offers a far more flexible method of scrolling the screen. Indeed, many professional AMOS coders now use software scrolling in preference to hardware scrolling simply because of



**Software  
scrolling**



its flexibility. It may not be quite as fast but, if used correctly, even software scrolling can turn in some very acceptable scroll speeds.



### SCROLL & DEF SCROLL commands

Software scrolling can be achieved using one of two different methods. By far the easiest method of software scrolling a section of a screen is to use the ‘Scroll’ command. However, before you can use this command you have to tell AMOS which section of the screen is to be scrolled using the ‘Def Scroll’ command. AMOS enables you to define up to 16 different scroll ‘zones’, each of which must be defined first using the Def Scroll command. The format of this command is as follows:

---

**Def Scroll N,X1,Y1 To X2,Y2,DX,DY**



**N** The parameter ‘N’ is an index number which is used to identify the scroll zone that you are defining. Up to 16 scroll zones can be defined simply by passing a value between 0 and 15.

**X1/Y1** X1 and Y1 define the top left-hand corner of the scroll zone in terms of screen pixels. If you wanted your scroll ‘zone’ to start at 10 pixels across and 20 pixels down, you would pass values of 10 and 20 respectively.

**X2/Y2** X2 and Y2 define the bottom right-hand corner of the scroll zone.

**DX/DY** DX/DY define the number of pixels the zone will be scrolled in a single operation. Positive numbers in DX and DY will scroll the zone to the right and down whilst negative values will cause the zone to be scrolled to the left and up.

DX and DY don’t actually tell AMOS where the scroll zone is to be placed on the destination screen, however – they are actually nothing more than ‘delta’ (relative) values that define the direction and speed of movement. Placing a value of -1 into DX, for example, would cause the scroll zone to be scrolled 1 pixel to the left every time the ‘Scroll’ command is called.

## Now use the Scroll command...

The 'Def Scroll' command won't do a lot on its own. To make the screen scroll, you need to use the 'Scroll' command. Its format is simple as it needs only one parameter – 'Scroll N' – where 'N' is the index number of the scroll zone that we defined using the 'Def Scroll' command. The scroll command is really nothing more than a block copier that uses the Amiga's blitter to cut out a rectangular section of the screen and paste it down at a given position. The command keeps track of how far the scroll has progressed, so there's no need to use any sort of counter to fix the current scroll position. One limitation of the scroll command is the way it leaves a trail as the rectangular zone is scrolled, so you have to redraw the corrupted sections of the screen. Here's a short demonstration:



```
Rem *** Software Scrolling Demo using 'Scroll' command
Rem *** Filename - Scrolldemo.AMOS
```



```
Screen Open 0,320,256,32,Lowres
Flash Off : Curs Off : Cls 0
```

```
Load Iff "AMOSBOOK:Pictures/DemoPicture.IFF"
Locate 0,16
Pen 31 : Paper 0 : Centre "Press Left Mouse Button to Quit!"
Double Buffer
AutoBack 0
```

```
Def Scroll 1,20,10 To 300,110,1,0
Def Scroll 2,20,150 To 300,250,-1,0
```

```
Repeat
    Scroll 1
    Scroll 2
    Wait Vbl
    Screen Swap 0
Until Mouse Key
```

```
Screen Close 0
End
```

## Screen Copy scrolling

A more flexible method of scrolling the screen under software control is to use the AMOS 'Screen Copy' command, which is now the preferred method of software scrolling screens amongst most professional AMOS programmers. With a little bit of clever coding, it can be made to scroll the screen smoothly at quite an acceptable rate and you can even create some quite complex 'parallax' scrolling effects with it.

The Screen Copy command was primarily designed as a quick and easy method of copying the contents of a rectangular region from one screen to another using the Amiga's blitter. Because the Screen Copy command makes use of the blitter, it is very fast indeed. It isn't just restricted to simple cut and paste operations, however – it's also a very capable scroll command too! The format of the Screen Copy command is as follows:

---

Screen Copy SOURCE, X1, Y1, X2, Y2 To DESTINATION, DX, DY, MODE



**Source/Destination** These two parameters define the two screens to be used for the Screen Copy operation. The 'Source' parameter tells AMOS which screen the rectangular scroll area is to be cut from and 'Destination' defines the area to be pasted into. The Screen Copy command can be used to transfer rectangular sections of one screen to another – but quoting the same screen number for both source and destination screens lets you scroll a section on the same screen.

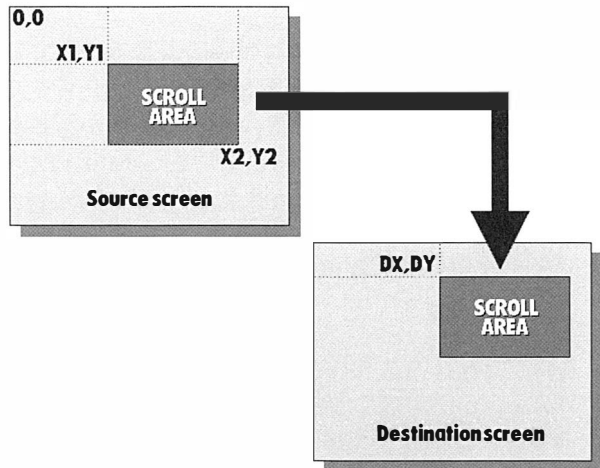
Say, for example, you were writing a game which required a small area of the screen to be continuously scrolled to the right or left (Defender, for example). Using a separate screen that is hidden from view (using the Screen Hide command), you could have the entire scroll area drawn into a single screen as a series of rectangular strips that could be cut out and pasted into the visible screen using the Screen Copy command.

**X1,Y1,X2,Y2** The X1,Y1 and X2,Y2 parameters defines the size and location of the rectangle (in screen pixels) that the Screen Copy command cuts out relative to the top left hand corner of the screen (screen position 0,0).



**Smooth  
scrolling**

The Screen Copy command may not be designed for scrolling, but it can be used to great effect to produce smooth screen scrolling effects.



**DX,DY** The DX and DY parameter define where the rectangular region cut from the source screen is to be placed within the destination screen. AMOS places the top left hand corner of the region at these co-ordinates on the destination screen.

**Mode** The mode parameter (which is optional) isn't particularly applicable to software scrolling, but let's take a look at it nonetheless. What it does is to allow you to take advantage of the blitter's ability to perform logic operations on an area of screen memory as it is being transferred. The values that you pass are called 'Minterms' and they're in the same format as assembler programmers use. The default value is %11000000 (this is a binary number) which just copies the block 'as is', but some interesting results can be achieved by altering this value. If you're feeling adventurous, why not try values of %00110000, %11110000 and %01010000, for example? You can find out more about Minterms in the Commodore 'Hardware Reference Manual' (published by Addison Wesley).

```
Rem *** Software Scrolling Demo using 'ScreenCopy' command
Rem *** Filename - ScreenCopy.AMOS
```



```
Screen Open 0,320,256,32,Lowres
```

```
Flash Off : Curs Off : Cls 0

Load Iff "AMOSBOOK:Pictures/DemoPicture.IFF"
Locate 0,16
Pen 31 : Paper 0 : Centre "Press Left Mouse Button to Quit!"

Do
  For C=20 To 200
    Screen Copy 0,C,10,C+100,110 To 0,110,150
    Wait Vbl
    If Mouse Key Then End
  Next C
  For C=200 To 20 Step -1
    Screen Copy 0,C,10,C+100,110 To 0,110,150
    Wait Vbl
    If Mouse Key Then End
  Next C
Loop

Screen Close 0
End
```

---

## Parallax scrolling

So far, the scrolling routines that we've covered have only scrolled screens at a fixed rate. Whilst they still look damned impressive, these days this sort of thing is starting to look decidedly old hat. If you own even a half decent shoot 'em up such as the classic 'StarRay' (remember that one?) or an arcade exploration game such as Psygnosis' 'Shadow of the Beast', then you may have noticed that the scroll routines used by these games manage to give a far greater illusion of 3D depth to the screen. There's nothing magical about this effect and, although it is a little more complex, there's no reason why you can't achieve similar results with AMOS. This technique is called 'parallax' scrolling and it works simply by scrolling certain areas of the screen at different speeds.

Parallax scrolling is a simple yet very effective scrolling technique that attempts to emulate the way we see moving objects in the real world. As

your physics teacher no doubt tried to drum into you back in your schooldays, the rate at which an object appears to move is directly related to the distance that it is from the viewport – even if a group of objects are moving at the same speed, they will appear to move past you at different speeds depending upon how far away they are. If an object is close, then it will appear to move at high speed. If, however, the same object was further away from you, it would appear to move more slowly.

We can emulate this natural phenomenon within AMOS by scrolling sections of the screen at different speeds using a technique that has been used to great effect within many high-speed arcade games. Potentially the fastest method of producing parallax scrolling is to use the AMOS ‘Dual Playfield’ command. This enables you to overlay one screen on top of another to create a dual playfield display. What’s more, the background colour from the foreground screen is made transparent therefore allowing any graphics that are displayed on the background screen to show through the gaps in the first screen. You can create parallax effects quite simply by using the Amiga’s own hardware scrolling capability.



**DUAL  
PLAYFIELD  
command**

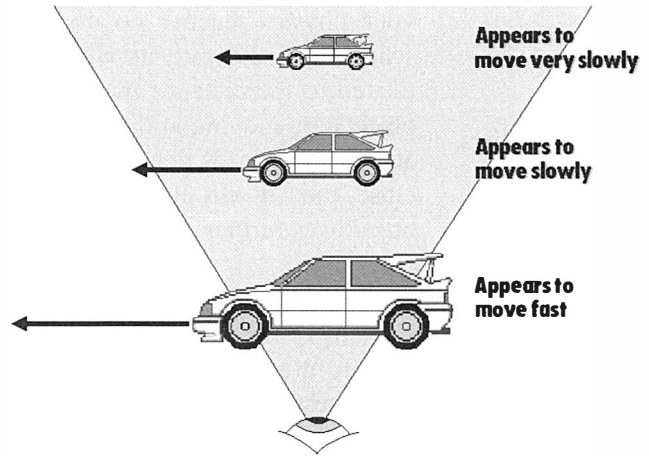
This may sound like just what we need, but the bad news is that dual playfield displays have a number of quite major limitations that make them virtually unusable. One is the number of colours that can be used – because you’re effectively combining two screens into one, the Amiga’s hardware only allows a maximum of sixteen to be displayed on each in low resolution mode (this drops to just eight when using medium and high resolution dual playfield displays!). What’s more, both screens have to be the same resolution (you cannot therefore have a low resolution screen overlaid on top of a medium resolution screen).

The greatest limitation of dual playfield screens is not imposed by the Amiga, but AMOS itself. Although it pains me to admit it, AMOS seems to have considerable problems with dual playfield screens – indeed, even opening a dual playfield display can sometimes produce weird (and not so wonderful) effects (corruption), so don’t blame yourself if you can’t get dual playfield mode to work correctly – the chances are that AMOS is to blame! The best way to avoid such problems is to open the two screens that you wish to combine using odd index numbers (screen 1 and

Objects that are further away appear to move more slowly – this is the basic theory behind parallax scrolling.



### Parallax scrolling



### Using 'dual playfields'

3, for example) and don't attempt to do horizontal hardware scrolling! Vertical scrolling usually works OK, but even this sometimes produces weird results. Just for the sake of an example, however, here's a listing that demonstrates dual playfield displays and the problems you can encounter – keep your eyes open for the juddering background! – you can blame AMOS for this:

```
Rem *** Dual Playfield Parallax demo
Rem *** Filename - DualPlayfield.AMOS
```



```
Screen Open 1,640,256,4,Lowres
Flash Off : Curs Off : Cls 0
Screen Open 3,640,256,4,Lowres
Flash Off : Curs Off : Cls 0
Screen Display 1,128,42,320,256
Screen Display 3,128,42,320,256
```

```
Screen 1 : Load Iff "AMOSBOOK:Pictures/Foreground.IFF"
Screen 3 : Load Iff "AMOSBOOK:Pictures/Background.IFF"
Screen Copy 1,0,0,320,256 To 0,320,0
Screen Copy 3,0,0,320,256 To 0,320,0
```

```

Wait Vbl
Dual Playfield 1,3

PF1POS=0
PF2POS=0

Repeat
  PF1POS=PF1POS+2
  PF2POS=PF2POS+1

  If PF1POS=320 Then PF1POS=0
  If PF2POS=320 Then PF2POS=0

  Screen Offset 1,PF1POS,0
  Screen Offset 3,PF2POS,0
  Wait Vbl
Until Mouse Key

Screen Close 1
Screen Close 3
End

```



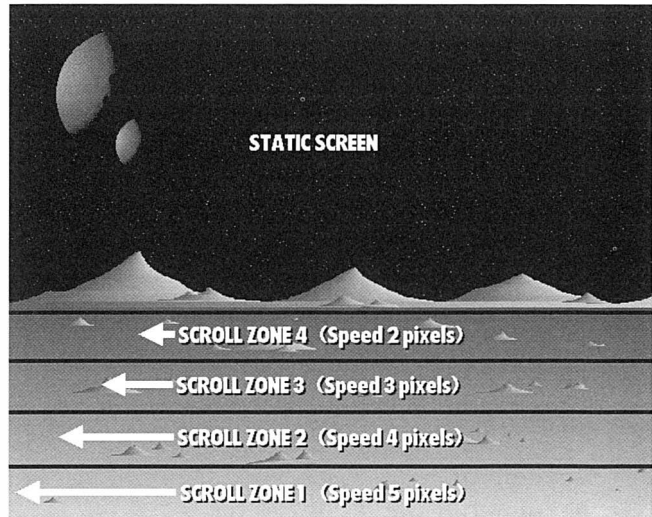
**SCREEN COPY**  
command

A far safer (and considerably easier) method of producing a dual playfield display is to make use of our old friend, the ‘Screen Copy’ command. It’s not quite as impressive as a true dual playfield display (you cannot, for example, have sections of one scroll area showing through another), but it’s considerably faster and a lot more flexible than dual playfield mode. Another great advantage of using Screen Copy to produce your parallax scroll is that it doesn’t restrict you to just two scrolling areas like a dual playfield parallax scroll. Finally – and rather importantly – the great thing about using Screen Copy is that it actually works – something that most certainly can’t be said of a dual playfield scroll!

As I’ve already said when we looked at the Screen Copy command earlier, it allows you to copy large areas of the screen using the Amiga’s fabulous blitter chip. As a result, it’s very, very fast indeed. The Screen Copy command can be used to produce a parallax scroll by splitting the



By scroll certain areas of the screen at different speeds, an illusion of true 3D depth can be added to your scrolling screens. This technique is called 'Parallax' scrolling.



screen into a series of bands, each of which is scrolled at a different speed using a separate 'Screen Copy' command. By arranging these scroll bands so that they move progressively faster (band 1 moves at a rate of 1 pixel, band 2 at 2 pixels and band 3 at 3 pixels etc), a very sexy looking parallax effect can easily be achieved. Here's a listing that demonstrates the Screen Copy command in action:

```
Rem *** Parallax Demo Using 'Screen Copy'
Rem *** Filename - Parallax.AMOS
```



```
Dim POS(5)
Global POS()

_INITSCROLL
Do
  _PARASCROLL
  Screen Swap 0
  Wait Vbl
Loop
```

## Procedure \_INITSCROLL

```
Screen Open 7,640,200,8,Lowres
Flash Off : Curs Off : Cls 0
```

```
Load If "AMOSBOOK:Pictures/ParallaxBackground.IFF"
Screen Copy 7,0,0,320,200 To 7,320,0
Screen Hide 7
```

```
Screen Open 0,320,256,8,Lowres
Flash Off : Curs Off : Cls 0
Get Palette 7
Double Buffer : Autoback 0
```

```
For C=0 To 4
  POS(C)=0
Next C
```

```
End Proc
```

## Procedure \_PARASCROLL

```
For C=0 To 3
  Screen Copy 7,POS(C),16*(C),POS(C)+320,16*(C+1) To
  0,0,100+(16*(C+1))
  POS(C)=POS(C)+C+2
  If POS(C)>=320
    POS(C)=0
  End If
Next C
```

```
Screen Copy 7,POS(4),64,POS(4)+320,96 To 0,0,84
POS(4)=POS(4)+1
If POS(4)>320
  POS(4)=0
End If
```

```
End Proc
```

## Continuous scrolling

The only real problem with both the hardware and software scrolling techniques that we've covered so far is that they work on a very limited scroll area. Take hardware scrolling, for example – because the scroll wraps around when you reach the end of the bitmap, the same background graphics will be shown over and over again. Unless your game is based in a barren desert of out in deepest space (so deep in fact, that there's nothing but stars around you!), this isn't particularly useful.

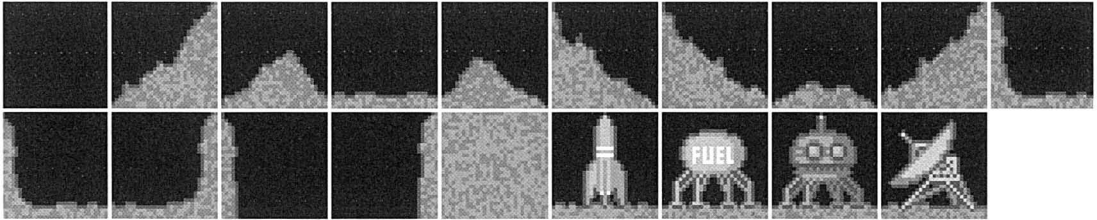
As any seasoned games player will acknowledge, most modern arcade games don't just scroll the same background screen over and over again. Take a game such as Psygnosis's 'Blood Money', for example – although the game will happily run on a 512K Amiga, its four levels are absolutely massive and as you progress through the game, the background graphics seem to change continuously. If you were to hold each level as a continuous bitmap, you'd end up needing a machine equipped with several megabytes (all of which was chip RAM!) of memory just to handle the background bitmaps.

So how do professional games programmers manage to squeeze so much into so little? Well, they use a little software trick that has been around for almost as long as home computers themselves. They break the bitmap down into a series of tiny 'blocks', each of which is a regular size (16 x 16 or 32 x 32 pixel blocks are most common). If you look carefully at the bitmap, you'll notice that quite a few of the blocks are very similar – a wall, for example, would use a same pattern of bricks that would not change between different blocks. You could therefore cut down on the amount of memory used simply by using the same block again and again.



**Screen 'blocks'**

Virtually all arcade games that feature continually changing scrolling backdrops use pretty much the same technique. Instead of holding the entire bitmap in memory, a small screen-sized area of the scrolling area is drawn by building up the display from a set of graphic blocks. As the screen is scrolled, the game interrogates a 'map' that holds nothing more than the numbers of the blocks that should be placed at certain positions on the screen. By using this data, only the areas of the screen that can actually be seen are ever drawn. What's more, because we use the



The entire scrolling screen area is built up using just a small selection of screen blocks. Using this technique, massive scrolling backdrops can easily be used without swallowing large chunks of memory.

Amiga's super-fast blitter, we can continuously draw the bitmap as the game progresses without slowing the game down at all!



**Memory savings**

Holding the screen as a 'map' offers considerable memory savings. The demo listing below, for example, creates a scroll area that is effectively 1280 by 200 pixels in size yet the entire program (which, incidentally, is on the disk bundled with this book) is under 20K in size (including the graphics themselves). When you consider that this same scroll area held as a continuous bitmap would eat up 128K, you can see that we've saved ourselves a lot of memory.

The great thing about this routine is that no matter how much you extend each screen, the amount of memory used will only increase by a couple of K (each 320 by 256 screen eats up only 80 bytes!). You could, for example, have a screen that was over 12800 pixels in size (that's ten times the size of the original) and the amount of extra memory used would only increase by a matter of K!

Our AMOS version of this routine works using a combination of hardware scrolling and the 'Screen Copy' command which is used to draw the blocks onto the screen very quickly indeed. The best way to explain how it works is to break the process down into a series of steps (see the diagram over the page):

**Step A** Before the scroll begins, the initial screen area is initialised by drawing the visible screen area, the scroll area and the two boundaries on either side of the viewport. Note how the screen viewport is initially offset so that there is a invisible area to the left of the screen – this will eventually

be displayed when the scroll routine reaches step 'F'. To start the ball rolling, strip 5 is drawn into it.

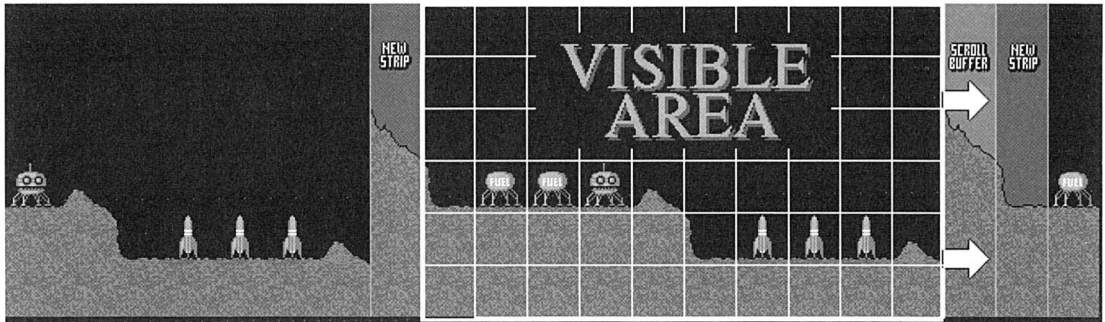
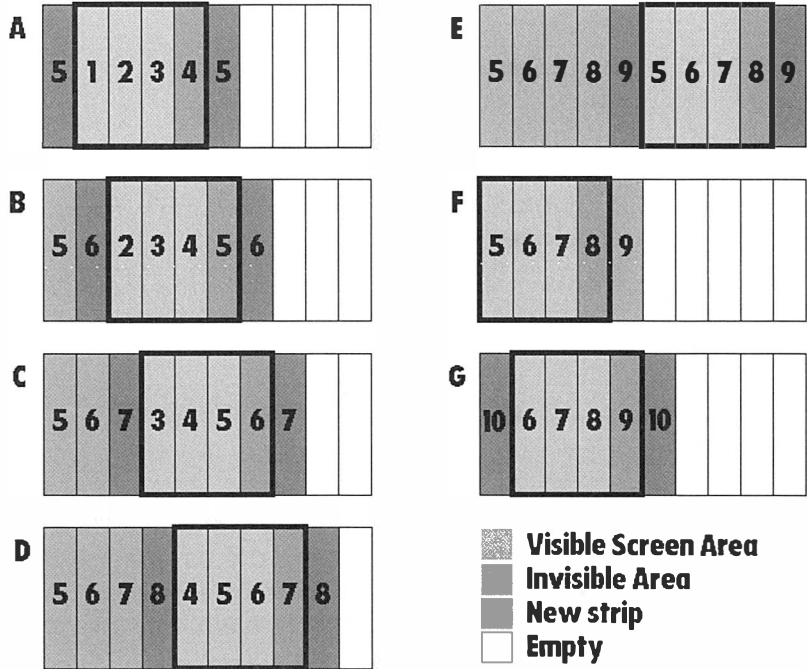
- Step B** The screen is then hardware-scrolled, and while it scrolls to reveal strip 4, strip 6 is drawn on either side of the viewport. All the icons that must be drawn to complete both strips are drawn at once to keep the scroll speed up.
- Step C** While the scroll moves on to reveal strip 5, strip 7 is drawn on either side of the viewport. Note that we will effectively need two pointers to keep this redraw process up – one that keeps track of the screen offset position and another that keeps track of which strip is to be drawn during each pass of the scroll routine.
- Step D** While the scroll moves to reveal strip 6, strip 8 is drawn on either side of the viewport.
- Step E** Again, the viewport is scrolled to the right and strip 9 is drawn on either side.
- Step F** Looking at the diagram opposite, you will now notice that the screen contains two copies of the same display which, in turn, is split into five strips – 5, 6, 7, 8 and 9. At this point, the viewport wraps around to the start so that it now displays the first half of the bitmap.
- Step G** With the viewport position reset, the whole scrolling process starts again by scrolling the viewport to display strip 9 whilst drawing strip 10 onto either side of the viewport. Once this is done, the scroll routine then jumps to step 'B' and the whole process begins again.

As I have already said, the actual layout of the screen is held as nothing more than a series numbers that define which blocks are to be pasted on the screen at certain positions. But how do you go about designing such a monolithic display? Well, I find the best way is to use a paint package such as DPaint. Simply by splitting the scroll area up into a series of smaller chunks (DPaint, for example, will quite happily handle a 1280 by 512 bitmap), you can design the screen by pasting down your blocks using DPaint's 'Grid' tool. Once one section is complete, you can

convert it manually into a series of block numbers that can be fed directly into your program. Drawing a grid over the finished bitmap makes life easier here.



**Scrolling**



If you were to view the entire bitmap as it was being scrolled, this is the sort of thing you'd see. Note how the entire image is built up using little graphic blocks.

If you do use this technique, make sure that the sections of screen that you design link together without a visible seam – as any games programmer will tell you, there’s nothing tackier than being able to spot that the display was drawn using screen blocks. Anyway, here’s the code for your perusal:

```

Rem *** Continuous screen scroll
Rem *** Filename - ContinuousScroll.AMOS

Dim LEVELMAP(6,80)
SCRPOINTER=0 : BLOCKPOINTER=0 : LEVELSIZE=0
BLOCKNUM=0 : SCRLCOUNTER=0
Global SCRPOINTER,BLOCKPOINTER,LEVELMAP(),LEVELSIZE
Global XDATA,BLOCKNUM,SCRLCOUNTER,X,Y,BLOCK

_INIT
_INITSCROLL

Repeat
  Rem *** Scroll screen...
  _SCROLLSCREEN

  Rem *** Rest of your game code goes here...

  Wait Vbl
Until BLOCKPOINTER=LEVELSIZE

End

Rem *** Initialise screen map array...
Procedure _INIT

  Rem *** Read length of screen in screen blocks...
  Read LEVELSIZE

  Rem *** Read level data into array...
  For A=0 To 5

```



```
      For B=0 To LEVELSIZE
        Read LEVELMAP(A,B)
      Next B
Next A

Rem *** Level 1 (1280x200) data
Data 39
STRIP1:
Data 1,1,1,1,1,1,1,1,1,1
Data 1,1,1,1,1,1,1,1,1,1
Data 1,1,1,1,1,1,1,1,1,1
Data 1,1,1,1,1,1,1,1,1,1
STRIP2:
Data 1,1,1,1,1,1,1,1,1,1
Data 1,1,1,1,1,1,1,1,1,1
Data 1,1,1,3,15,15,15,5,1,1
Data 1,1,1,1,1,1,1,1,1,1
STRIP3:
Data 1,1,1,1,1,1,1,1,1,1
Data 1,1,1,1,1,1,1,1,1,1
Data 1,1,2,14,14,14,14,14,6,1
Data 1,1,1,1,1,1,1,1,1,1
STRIP4:
Data 1,1,1,1,3,18,15,15,5,1
Data 1,1,1,1,1,1,1,1,1,1
Data 1,2,14,14,14,14,14,14,14,10
Data 16,16,17,5,1,1,1,1,1,1
STRIP5:
Data 1,1,1,2,14,14,14,14,14,7
Data 16,5,1,1,1,1,1,1,3,18
Data 9,14,14,14,14,14,14,14,14,14
Data 14,14,14,14,10,15,15,15,5,1
STRIP6:
Data 4,4,9,14,14,14,14,14,14,14
Data 14,14,7,4,17,15,15,9,14,14
Data 14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14
Data 14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,7
```



End Proc

Procedure \_INITSCROLL

Rem \*\*\* Open up invisible bitmap for screen blocks

Screen Open 7,640,40,16,Lowres

Flash Off : Curs Off : Cls 0

Rem \*\*\* Load in screen blocks...

Load Iff "AMOSBOOK:Pictures/ScrollBlocks.IFF"

Screen Hide 7

Rem \*\*\* Open up scroll bitmap...

Screen Open 0,736,192,16,Lowres

Flash Off : Curs Off : Cls 0

Screen Display 0,128,48,320,200

Screen Offset 0,32,0

Get Palette 7

Screen Hide 0

Rem \*\*\* Initialise screen display

SCRPOINTER=32

XDATA=0

BLOCKPOINTER=XDATA+10

Rem \*\*\* Draw main display + scroll buffer...

For A=0 To 5

For B=0 To 9

BLOCK=LEVELMAP(A,XDATA+B)

X=SCRPOINTER+(B\*32)

Screen Copy 7,(BLOCK-1)\*32,0,BLOCK\*32,32 To 0,X,A\*32

Next B

Next A

Rem \*\*\* Draw new strips...

BSTRIP=SCRPOINTER-32

FSTRIP=SCRPOINTER\*11

For A=0 To 5

```
BLOCK=LEVELMAP(A,BLOCKPOINTER)
Screen Copy 7,(BLOCK-1)*32,0,BLOCK*32,32 To 0,BSTRIP,A*32
Screen Copy 7,(BLOCK-1)*32,0,BLOCK*32,32 To 0,FSTRIP,A*32
Next A

Rem *** Bring scroll bitmap into view...
Screen Show 0
End Proc

Procedure _SCROLLSCREEN

Rem *** Reset scroll pointer if edge
Rem *** of screen bitmap has been reached...
If SCRPOINTER=352 Then SCRPOINTER=0

Rem *** Draw new strips...
If SCRLCOUNTER<12
  If BLOCKNUM<6
    X=SCRPOINTER+320
    Y=BLOCKNUM*32
    BLOCK=LEVELMAP(BLOCKNUM,BLOCKPOINTER)
  Else
    X=SCRPOINTER-32
    Y=(BLOCKNUM-6)*32
    BLOCK=LEVELMAP(BLOCKNUM-6,BLOCKPOINTER)
  End If
  Screen Copy 7,(BLOCK-1)*32,0,BLOCK*32,32 To 0,X,Y
  Inc BLOCKNUM
Else
End If

Rem *** Scroll bitmap...
Screen Offset 0,SCRPOINTER+SCRLCOUNTER,0

Rem *** Reset redraw delay if screen has scrolled 32 times
Inc SCRLCOUNTER
If SCRLCOUNTER=32
```

```
SCRLCOUNTER=0
SCRPOINTER=SCRPOINTER+32
Inc BLOCKPOINTER
BLOCKNUM=0
End If
End Proc
```

---

## AMOS TOME Extension

If you own AMOS 1.35, you can cut out a lot of the hassle of scrolling large continuous bitmaps by making use of Shadow Software's excellent 'Tome', an AMOS extension designed specifically for handling 'map'-based games. Now in its forth incarnation (although even as I write this, Tome Series V is on the drawing board!), Tome gives you an extra 60 commands designed specifically for the task of handling big scroll areas. The only real problem with this extension is the fact that it will not work with either Easy AMOS or even AMOS Professional, so I'm afraid the rest of us will have to make do with the code above.



### Tome support

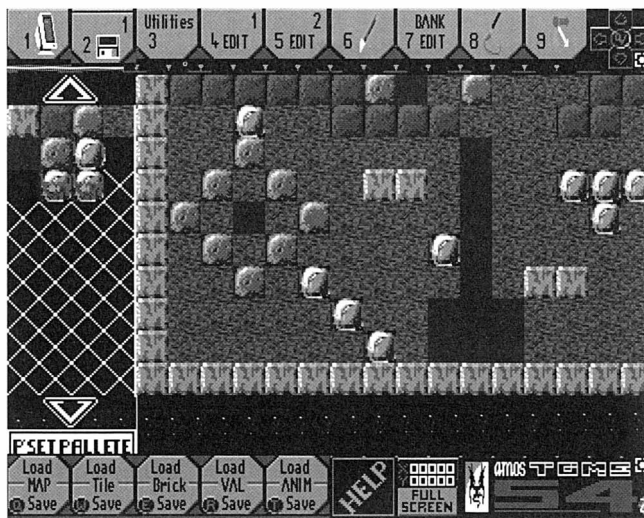
The Tome package consists of primarily two basic elements – the Tome language extension and a Map Editor utility that is used to design your scrolling backgrounds. Tome works in pretty much the same way as our continuous scrolling routine above but, thanks mainly to the fact that the Tome commands are written completely in assembler, it can redraw the background much, much faster than our Screen Copy-based routine. The first step is to design a series of 'Tiles' (Blocks) that will form the building blocks of the map display. Tome allows you to use just about any paint package for this, but DPaint is perhaps better qualified than most. Once you've done this, you can then load up the Tome Map Editor, cut out your blocks from the IFF file created by your paint package and then piece together these tiles into a massive scrolling area.

The latest release of Tome (series IV) includes a couple of extra utilities that are built into the Tome Map Editor that make designing maps easy. The most impressive of these is a small utility called 'Picture To Map Converter' that takes an IFF picture and produces a massive map by converting each pixel in the picture into a Tome tile. This comes in particularly handy if you're designing games that use a lot of very similar

The Tome Map Editor allows you to design enormous scrolling backgrounds that can then be incorporated into your own games software using the 60+ commands in the Tome extension.



**AMOS Tome**



blocks – an arcade adventure based in a wilderness, for example. To accelerate the process of designing a map for this sort of game, all you would have to do is to draw a quick picture with each type of tile represented by a particular colour. Tome Series IV also supports ‘animated tiles, so you can animate sections of a scrolling background!

Once the map is designed, a few quick ‘Load’ commands are all that is needed to pull your map into your AMOS program. From here on you can use the 60 or so Tome commands to automatically redraw the screen map to create the illusion of scrolling. Here’s a rundown of a few of Tome’s obvious commands just to show you how easy it is to use.

### Map Do X,Y

This command draws an area of the map into the current screen. Tome starts drawing the map at the top left-hand corner of the screen using the tile at position X and Y within the map.

### Map View X1,Y1 To X2,Y2

If you need to limit the size of the visible map area (say, for example, you’d like to leave the bottom of the screen free for a status panel), you can use this command to create a sort of ‘window’ which restricts the size of the screen area that the ‘Map Do’ command draws the map into.

**Map Left X,Y**  
**Map Right X,Y**  
**Map Top X,Y**  
**Map Bottom X,Y**

These four commands are used in conjunction with your screen scrolling code to produce smooth scrolling. All you do is to scroll the screen and use one of these four commands to fill in the gap left by the screen being scrolled.

**Map Plot TILE,X,Y**

This command draws tile number 'TILE' at position X,Y within the map. Note that these two co-ordinates are not screen-based – passing a value of 12,20, for example, would plot a tile at position 12 x 20 within the map (even if that position can't currently be seen!).

**Map Fall TILE**

If you've ever wanted to write your own 'BoulderDash' clone, then this is the command for you. What it basically does is to scan through the map, causing certain tiles to fall downwards. It's a bit complicated to explain how it works, but it works very well indeed!

The AMOS Tome extension costs £24.95 and is available from Shadow Software at 1 Lower Moor, Whiddon Valley, Barnstaple, North Devon EX32 8NW. Bear in mind that, as stated earlier, at the time of writing, Tome is not compatible with either AMOS Professional or Easy AMOS.

# Screen effects

- The Amiga's co-processor ('copper')
- Copper effects and rainbows
- Screen synchronisation & 'double-buffering'
- Screen compaction



**Denise/Lisa,  
Agnus/Alice,  
copper**

**A**t the heart of every Amiga is a wondrous little sliver of silicon called ‘Denise’ (or ‘Lisa’, if you have an AGA machine) that is solely responsible for translating the screen bitmaps generated by the Amiga’s graphics chip ‘Agnus’ (‘Alice’ on AGA systems) into the displays that you see on your Amiga’s monitor or television. Built into the heart of Denise is a sort of mini co-processor called the ‘copper’ (short for ‘co-processor’) that allows the Amiga to change certain hardware registers (a screen’s colour palette, for example) every scan line. By creating what the techies call a ‘copper list’, all manner of weird and wonderful screen effects are possible.



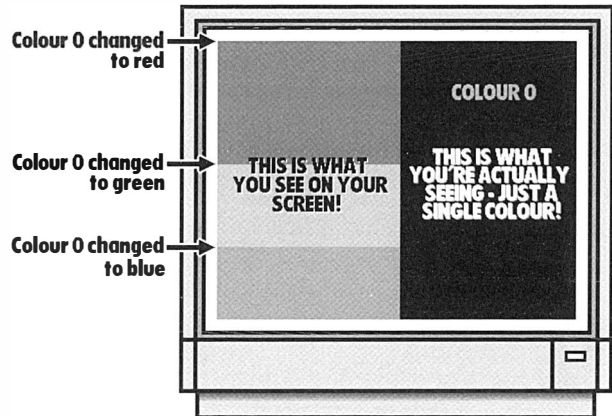
**Colour  
limitations**

Most games programmers use the copper extensively simply because it allows you to beat the 32-colour limitation imposed by the Amiga’s graphics hardware. As you will know, AMOS allows us to use no more than 32 colours on any screen. You could, of course, make use of HAM mode, but this rather quirky screen mode is more bother than it’s worth. What’s more, HAM mode doesn’t like blitter objects (bobs), so it’s not a lot of use for game screens. The only other solution is the AGA chip set, which breaks these barriers completely with its new 262,000-colour and 256-colour VGA screen modes, but until Europress launch an AGA upgrade for AMOS, I’m afraid no amount of creative coding is going to allow AMOS users to take advantage of these wondrous screen modes. Let’s just hope that the AGA upgrade doesn’t take too long to arrive!

*The Amiga’s copper chip allows the AMOS programmer to change the setting of a colour register anywhere on the screen, giving the illusion of many more colours.*



**Using the  
copper!**



This then is where the copper comes in handy. Because it allows us to change the colour of any colour register at any point on the screen, there's no reason whatsoever why you couldn't have the same colour register (colour 0, for example) displaying several different colours on the same screen. You could, for example, have colour 0 set to black at the top of the screen and then change it to red half way down the screen!

Compared to the power of the copper, even this is a very simple example of what's possible – if you wanted to, you could change its colour on every scan line to create beautiful multi-colour effects using just a single colour! Don't believe me? Try typing in this example listing:

```
Rem *** Rainbow Effect Demo
Rem *** Filename - RainbowDemo.AMOS

Screen Open 0,320,256,2,Lowres
Flash Off : Curs Off : Cls 0

Set Rainbow 0,0,280,"(1,1,15)","",""
Rainbow 0,0,0,280
End
```



When you run this listing, you should see a screen filled with beautifully shaded red colour bars that fade gradually between black and bright red. OK, so it's nothing special, but hang on a second – study the listing and you'll see that the screen that we open at the start only uses 2 colours. So where are all those extra colours coming from?

Well, they're actually the same colour! All we've done is to create a copper list that changes the value of colour 0 every scan line so that it appears that we actually have loads of different shades of red. But, believe it or not, what you are actually looking at is nothing more than a completely blank screen!

AMOS calls its copper lists 'rainbows' and there's a whole range of commands that allow you to create some quite exciting copper effects.



---

## Defining a Rainbow

AMOS lets us create ‘copper lists’ using three commands – ‘Set Rainbow’, ‘Rainbow’ and ‘Rain()’. Before you can display a rainbow effect, however, you need to initialise your copper list using the ‘Set Rainbow’ command. The format of this command is as follows:

---

Set Rainbow **NUMBER**, **REGISTER**, **SIZE**, **RED**, **GREEN**, **BLUE**



**NUMBER** This first parameter is a number between 0 and 3 that defines the identifier number of the copper list. AMOS allows us to create up to four different copper lists, each of which must have its own unique identifier number.

**REGISTER** The register parameter tells AMOS which colour register is to be affected by your copper list. If you open a low resolution screen with 32 colours, then you can attach your copper list to any one of these colours. If you were to place a value of ‘0’ here, for example, then colour 0 (the background colour) would be affected. Place a value of ‘1’ and colour 1 is affected and so on.

Unlike real copper lists, only a single colour register can be effected by each copper list that you define, so only four colour registers can be changed by creating four separate copper lists.

**SIZE** The size parameter defines the size of your copper list in scan lines. This parameter therefore tells AMOS how many scan lines are to be affected by your copper list so, even if you wish to change a colour register only once, the size of your copper list must be big enough to reach down to where the change is to take place.

Say, for example, you wanted to change the background to red on scan line 1 and then change it to green on scan line 200. Even though you’re only making two changes, the total length of the copper list would have to be set to 200 so that all the scan lines between the start and finish of your copper list are covered.

**RED/GREEN/BLUE** These three strings allow you to create graduated fades from one colour to another. Each colour component comprises a string containing three values held within closed brackets. The format of each string is as follows:

(SCANLINES, STEP, COUNT)

**SCANLINES** The scanlines parameter defines how many scan lines will be effected by each colour change. If, for example, you wanted to fade between black and red in sixteen steps, sixteen scan lines would be required. You could expand this change, however, by increasing the value held in the 'scanlines' parameter. Passing a value of 1, for example, would make each colour change affect 1 scan line. A value of 4 though, would effectively quadruple the size of the change because four scan lines would be used for each new colour rather than 1. If you had set the size of the copper list to just 16, you would need to increase it to 64 so that all the scan lines would be covered.

**STEP** The step value allows you to specify a value that would be added to the colour setting each time that colour is changed. A value of 1, for example, would increase the colour by 1 and so on. Passing a negative value will cause the colour value to be decreased.

**COUNT** The count parameter controls how many times the value held in step is added to the colour register. It's a bit like a loop counter – if you pass a value of 4, for example, the colour would change 4 times.

---

## The Rainbow command

Once you've defined your rainbow, you can get AMOS to display it using the 'Rainbow' command. This command simply takes the copper list you created with the 'Set Rainbow' command and draws it onto the screen at the position that you specify. The format is as follows:

---

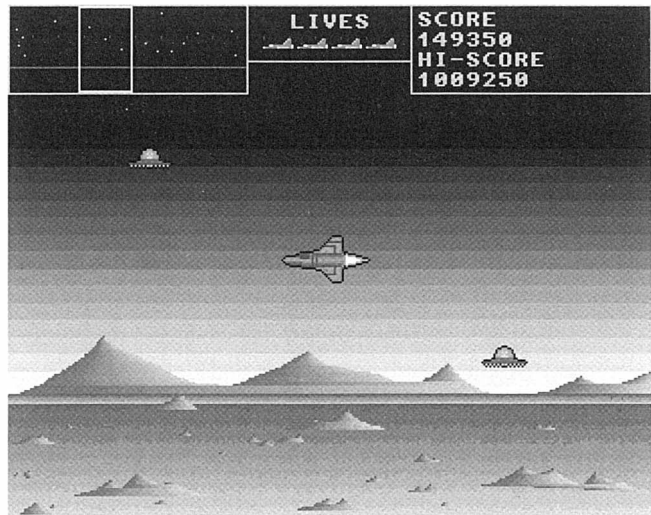
Rainbow NUMBER, OFFSET, POSITION, LENGTH



AMOS's powerful 'Rainbow' commands can be used to great effect to create wonderful shaded backgrounds for game screens.



**RAINBOW**  
command



**NUMBER** The number parameter simply tells the Rainbow command which copper list to use. This must be exactly the same value that you specified when you first created the copper list using the 'Set Rainbow' command. If you specify a copper list that hasn't been defined, you will get an 'out of memory' error.

**OFFSET** This sets an internal pointer telling AMOS where to start the copper list. It can be used to create animated rainbow effects.

**POSITION** The position defines where on the screen the copper list is to be placed. This value defines the start position of the copper list, so all colour changes are performed after this screen position. Note that this value isn't the same as the pixel co-ordinates used for drawing operations, though – it's what's known as a 'hardware' co-ordinate. As a result, only values between 40 and 298 are acceptable. To convert a pixel position into a hardware co-ordinate, use the AMOS 'Y Hard()' function.

**LENGTH** The length parameter tells AMOS how many scan lines the copper list is to cover. If you create a copper list that is only 16 scan lines in length, specifying a length larger than this will cause the copper list to be repeated. If you have a copper list that is 16 scan lines in length and you specify a length of 32 scan lines, for example, the copper list will

repeat twice. If you therefore want the copper list to be shown only once, then this value must be the same as the total length of the copper list. Here's a demonstration of the 'Set Rainbow' and 'Rainbow' commands:



```

Rem *** Rainbow Effect Demo 2
Rem *** Filename - RainbowDemo2.AMOS

Screen Open 0,640,256,2,HiRes
Flash Off : Cls 0

Input "Enter number of scan lines per colour change: ";SIZE
Print "Try moving the rainbow with the joystick!"

RED$="( "+Str$(SIZE)+" ,1,15)"

POSMIN=40 : POS=POSMIN
POSMAX=298-(SIZE*16)

Locate 1,4 : Print "Rainbow maximum position = ";POSMAX

Set Rainbow 0,0,16*SIZE,RED$,"",""
Rainbow 0,0,POS,16*SIZE
Curs Off

Repeat
  If Joy(1)=1 Then POS=POS-1
  If Joy(1)=2 Then POS=POS+1
  If POS<POSMIN Then POS=POSMIN
  If POS>POSMAX Then POS=POSMAX

  Locate 1,5 : Print "Rainbow position = ";POS;" "
  Locate 1,6 : Print "Rainbow size = ";16*SIZE

  Rainbow 0,0,POS,16*SIZE
  Wait Vbl
Until Joy(1)=16
End

```

## The Rain() function



### SET RAINBOW command

The 'Set Rainbow' command is fine for creating simple fades between one colour and another (usually black), but it does have its limitations. For starters, you're not given any form of real control over the colour assignment of individual scan lines, so it's hard to create anything other than the simplest of copper effects. Not only that, but you can only really produce sixteen colour changes which can only be placed at fixed widths between each other, so it's a little limiting to say the least. Say, for example, you wanted to change the background to white at position 10, change it to black at position 100 and then back to white at position 200. Using the 'Set Rainbow' command, this would be impossible.



### RAIN() function

This then is where the 'Rain()' function comes in. What it allows you to do is to change the colour setting of a scan line at any position within your copper list. Only a single scan line is affected, so the colour will revert back to its original value on the next scan line. If you wanted a colour to be set to a particular value across several scan lines, it's therefore up to you to change each and every scan line manually by calling the 'Rain()' function for each line to be affected. This isn't as involved as it sounds, however – all you need to do is to create a loop that fills in the gaps. The format of the 'Rain()' function is as follows:

`Rain(NUMBER, SCANLINE) = COLOURSETTING`



**NUMBER** Not surprisingly, this parameter simply tells AMOS which copper list we wish to modify...

**SCANLINE** The scanline parameter defines which scanline is to be affected by the change. A value of 200, for example, would tell AMOS that we want to change the colour setting for scan line 200.

**COLOURSETTING** This parameter defines the colour setting (expressed as a 3-digit hex value) to be placed into the copper list at the position defined by the 'Scanline' parameter. A value of \$000, for example, would place black into the copper list whilst a value of \$F00 would place red into the list.

The 'Rain()' function can also be used to read colour values held within a copper list simply by reversing the command. For example, the line 'Rain(0,200)=\$FFF' would change the colour affected by copper list 0 to white (\$FFF) at position 200 whilst the line 'RGB=Rain(0,200)' would copy the colour assignment at position 200 in copper list 0 to an AMOS variable called 'RGB'.

Here's a quick demonstration that shows the 'Rain()' function in action. All it does is to change the background colour 4 times on the screen. It's worth noting how the gaps between each colour change are automatically filled by a FOR...NEXT loop.

```
Rem *** Rain() function demo
Rem *** Filename - RainFunction.AMOS

Screen Open 0,320,256,2,Lowres
Flash Off : Curs Off : Cls 0

Set Rainbow 0,0,280,"", "", ""

Rem *** Fill first 40 scanlines with yellow
For C=0 To 40
    Rain(0,C)=$FF0
Next C

Rem *** Fill scanlines 41 to 120 with red
For C=41 To 120
    Rain(0,C)=$F00
Next C

Rem *** Fill scanlines 121 to 180 with green
For C=121 To 180
    Rain(0,C)=$F0
Next C

Rem *** Fill scanlines 181 to 279 with blue
For C=181 To 279
```



```

    Rain(0,C)=$F
Next C

Rainbow 0,0,0,280
End

```

You can also animate a copper list by altering the value held in the Rainbow command's 'offset' parameter. This can be used to great effect to produce the sort of 'copper bar' effects seen in many Amiga demos. Try typing in this short example program for a vivid demonstration:

```

Rem *** Animated Rainbow Effect
Rem *** Filename - AnimatedRainbow.AMOS

Screen Open 0,320,256,2,Lowres
Flash Off : Curs Off

Set Rainbow 0,0,192,"", "", ""

Rem *** Set up copper list
COUNT=0
For R=0 To 15
    RGB=Val(Hex$(R)+"00")
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next R
For R=15 To 0 Step -1
    RGB=Val(Hex$(R)+"00")
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next R
For G=0 To 15
    RGB=Val("$0"+Right$(Hex$(G),1)+"0")
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB

```



```
        COUNT=COUNT+2
Next G
For G=15 To 0 Step -1
    RGB=Val("$0"+Right$(Hex$(G),1)+"0")
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next G
For B=0 To 15
    RGB=Val("$00"+Right$(Hex$(B),1))
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next B
For B=15 To 0 Step -1
    RGB=Val("$00"+Right$(Hex$(B),1))
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next B

Rem *** Turn on and animated copper list
Do
    For C=0 To 191
        Rainbow 0,C,0,280
        Wait Vbl
    Next C
Loop
```

---

## Screen synchronisation

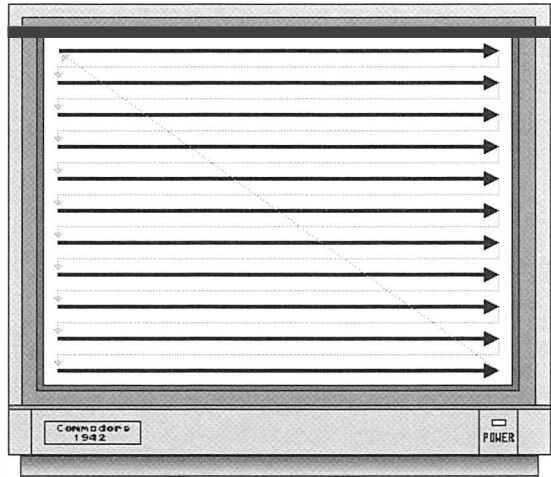
The Amiga's video hardware can shift blocks of graphics around the screen at a phenomenal rate, but there's one weak link in the chain that stops you from writing the sort of arcade game that would need three brains running in parallel just to keep track of the player's sprite – screen refresh. I briefly touched on this subject in the last chapter, but further coverage is definitely needed. Before we dive in too deeply, however, let's take a look at the theory behind screen synchronisation.





### Screen drawing

The Amiga's display is drawn onto your monitor or TV using an electron beam which scans from left to right down the screen 50 times a second. When it reaches the end, it is switched off and then reset to the top left hand corner.



As you will already know, the picture you see on your Amiga's monitor is redrawn every 50th of a second (or every 60th of a second on NTSC Amigas) using an electron beam (called the 'raster beam') which scans from the top left-hand corner of your monitor (or TV), drawing each horizontal line of the display as it goes. When the beam reaches the bottom right hand corner of the display, it is switched off and then reset to the top left-hand corner and the whole process then starts again. The time period during which the beam takes to move from the bottom right hand corner of the screen to its start position is called the 'vertical blanking period'. Obviously because the display is only redrawn 50 times a second, if your program redraws the screen any faster than this, strange things can happen. It's therefore best to redraw the screen during this vertical blanking period in order to avoid horrible flickers.



### WAIT VBL command

Most programs are based around a loop which will redraw the screen once each time the loop is performed. AMOS, being the super-slick programming language it is, can often perform several of these loops before the raster beam reaches the end of the display – OK, so your program will be running very fast indeed, but what's the point in redrawing the screen several times when your monitor can only redraw it once every 50th of a second? Thankfully AMOS has the answer with a handy little command called 'Wait Vbl' that, when placed inside a loop, forces the loop to be performed only once every 50th of a second. Wait

Vbl doesn't actually do anything though – all it does is to instruct your AMOS program to wait until the vertical blanking period before commencing any further drawing operations.

We used this command extensively in the last chapter to slow down many of the scroll routines – if we hadn't have used 'Wait Vbl', many of the scrolling demos would have scrolled the screen so fast that your monitor wouldn't have been able to keep up. Here's a demonstration program that shows screen synchronisation in action:

```

Rem *** Wait Vbl Demonstration
Rem *** Filename - WaitVBL.AMOS

Screen Open 0,320,256,32,Lowres
Flash Off : Curs Off : Cls 0

Load Iff "AMOSBOOK:Pictures/DemoPicture.IFF"

Pen 31 : Paper 0
Print "This is what happens when screen"
Print "synchronisation isn't used.."
Print "Now let's turn it on.. press space.."

COUNTER=10
Repeat
    Screen Copy 0,COUNTER,32,COUNTER+120,82 To 0,100,130
    COUNTER=COUNTER+1
    If COUNTER=200 Then COUNTER=0
Until Inkey$=""

Print "Now isn't that so much better!!!"

COUNTER=10
Repeat
    Screen Copy 0,COUNTER,32,COUNTER+120,82 To 0,100,130
    COUNTER=COUNTER+2
    If COUNTER=200 Then COUNTER=0

```



```
Rem *** Wait for vertical blanking period..
Wait Vbl
```

```
Until Inkey$="" "
End
```



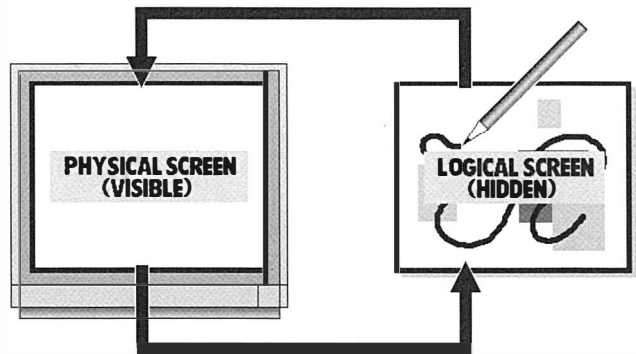
### 'Double Buffering'

If you're writing a particularly complex game or demo, there are times when even the fastest computers can't quite handle the task of updating a screen completely during the blanking period (yes, even an A4000/040 occasionally encounters speed problems!), so programmers use a system called 'Double Buffering' to ensure that screen redraw is kept silky smooth. Double buffering works by 'doubling up' a screen so that it actually consists of two separate bitmaps. Whilst one is being displayed on your monitor or TV screen, the other is used for all drawing operations. This hidden bitmap is called the 'logical' screen whilst the bitmap you can see on your TV or monitor is called the 'physical' screen.

Once all your drawing operations are complete (the screen has been scrolled and any bobs have been moved, for example), the physical and logical screens are swapped so that the logical screen is displayed on your monitor and the physical screen is hidden from view. At this point, the old physical screen becomes the new logical screen and vice-versa. You then perform all your drawing operations into the new logical screen and the process continues. Although double buffering doubles the amount of memory that a screen will eat up, the smoothness that it offers is well worth the memory penalties.

Setting up a double buffered screen is very easy indeed. All you have to do is to create the screen that you wish to double buffer (using the 'Screen Open' command) and then issue the 'Double Buffer' command and AMOS will automatically create the second logical bitmap for you. Whenever you call the 'WaitVbl' command, the physical and logical screens will automatically be swapped for you, so there's no extra work involved whatsoever. What's more, because the physical and logical bitmaps are treated as one and the same screen, you don't have to tell AMOS which bitmap is the logical screen and which is the physical – all this is done for you automatically!

Double buffering virtually removes all screen flicker by performing all drawing operations into a spare screen that is hidden from view. When all drawing operations are complete, the two screens are swapped as soon as the vertical blanking period is reached. This process is performed over and over again ensuring flicker free movement on screen.



There are – as always – a couple of minor problems with double buffering. For starters, AMOS does so much work for you that drawing into a double buffered screen is considerably slower than working with a normal screen. Once again, though, you can get around this limitation too by switching off AMOS’s AutoBack system by adding the line ‘AutoBack 0’ after you have created the double buffered display. Once this is done, AMOS heaps all the work of swapping screens onto your program – don’t worry, this isn’t as complicated as it sounds. All you have to do is to perform all your drawing operations and then simply call the ‘Screen Swap SCRNUMBER’ command. To be perfectly honest, automatic screen swapping is fine for very simple programs, but you’ll probably find it more trouble than it’s worth – most programmers these days switch it off almost instantly!



### Screen-swapping

The second problem with double buffered displays is that because you’re working with two bitmaps, any drawing operations that are carried out on the logical screen are not automatically transferred to the physical screen when the screens are swapped. It’s therefore down to you to ensure that both screens are updated accordingly. Say, for example, you copied a section of graphic into the logical screen – unless you copy it into the physical screen too, you’ll get a noticeable flicker because the Screen Copy command will have pasted the graphic into only one of the two bitmaps.

Once again, though, there is a way of getting around this too. Although both the physical and logical bitmaps are tied to the same screen, it’s still

possible to directly access either using a pair of very handy functions – ‘Physic(SCRNUMBER)’ and ‘Logic(SCRNUMBER)’. These two functions allow you to extract a unique identification number that points to the current physical and logical bitmaps. If, for example, you wanted to copy the entire contents of the physical screen to the logical screen using ‘Screen Copy’, all you’d have to do is to issue the following command.

Screen Copy Physic(0) To Logic(0)



It’s worth noting, however, that if you perform drawing operations onto a screen before the ‘Double Buffer’ command is called, the Double Buffer command will automatically copy the graphics into both the physical and logical screens. This can be handy if you need to load an IFF picture into a double buffered screen as a backdrop – all you have to do is to load the picture first and then turn on double buffering! Anyway, in time-honoured fashion, here’s an example listing that demonstrates double buffering in action. Type it in and have a play around with it – as they say, the best way to learn anything complicated is to experiment:

```
Rem *** Double Buffering Demonstration
Rem *** Filename - DoubleBuffer.AMOS
```



```
Global XPOS,YPOS
```

```
Screen Open 0,320,256,32,Lowres
Flash Off : Curs Off : Cls 0
Load Iff "AMOSBOOK:Pictures/DemoPicture.IFF"
Screen Hide 0
```

```
Rem *** Create a 'mask' around the area to be copied
Rem *** This effectively removes the old graphic
Rem *** when the graphic is moved.. clever eh!
Ink 0
Box 60,0 To 259,127
Box 61,1 To 258,126
```

```
Screen Open 1,320,256,32,Lowres
Flash Off : Curs Off : Cls 0
Get Palette 0

Pen 31 : Paper 0
Print "Here's a screen using just 'Wait Vbl'"
Print "Move the graphic around with a joystick"
Print "Press fire button to double buffer it!"

XPOS=50
YPOS=100

Repeat
    _CHECKJOYSTICK
    Screen Copy 0,60,0,260,128 To 1,XPOS,YPOS

    Wait Vbl
Until Joy(1)=16

Print "Now try moving the graphic!!!"

Double Buffer
Autoback 0

Rem *** Wait for fire button to be released..
Wait 10

Repeat
    _CHECKJOYSTICK
    Screen Copy 0,60,0,260,128 To 1,XPOS,YPOS

    Screen Swap 1
    Wait Vbl
Until Joy(1)=16
End

Procedure _CHECKJOYSTICK
```

```
If Joy(1)=1 Then YPOS=YPOS-1
If Joy(1)=2 Then YPOS=YPOS+1
If Joy(1)=4 Then XPOS=XPOS-1
If Joy(1)=8 Then XPOS=XPOS+1
If XPOS<0 Then XPOS=0
If XPOS>120 Then XPOS=120
If YPOS<40 Then YPOS=40
If YPOS>129 Then YPOS=129
```

End Proc

---

## Screen icons

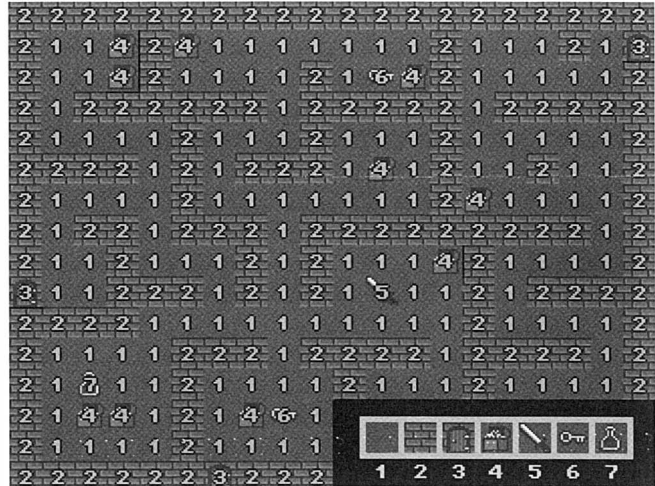
Conserving memory is all-important whenever you're coding games, especially if you want users with less than a megabyte of RAM to be able to run your software. As any games programmer will tell you, by far the most wasteful aspect of any game is its graphics – typically, they can eat up to 80% of the memory required to run an average game. It's unlikely that you'll encounter problems if your game uses just one or two screens, but how do you cram tens or even hundreds of different screens into memory simultaneously? Well, you could buy yourself an absolutely enormous RAM expansion, but how many other users do you think will have this sort of power at their disposal? Very few, I can assure you (most users have no more than 1 Mb).

A far better (and certainly much cheaper) method is to take advantage of a programming technique used by just about every games programmer these days. By splitting a screen up into a series of tiny blocks (often just 16 by 16 pixels in size), you simply build up each screen by pasting down and reusing the same blocks over and over again. It's a very similar technique to the screen redrawing routine we used in the 'Continuous Scroll' program covered in the last chapter. Because each screen is simply held as a series of numbers that define which blocks are to be pasted where, hundreds of screens can be packed into memory without having to worry about memory constraints.



The key to all this screen trickery is AMOS's 'Icon' commands, which allow you to paste down rectangular areas of graphics onto the screen in double-quick time using the Amiga's blitter chip. As a result, it's often

Hundreds of screens can be crammed into a game by building each screen up from a small set of graphic blocks held in an AMOS Icon Bank.



considerably faster to use this technique rather than to load each screen from disk as the player completes a level.

In order to make use of screen icons, you need to draw up each block in a paint program (I personally use DPaint) and then transfer them across into an AMOS 'Icon Bank' using the AMOS Object Editor accessory (we'll be taking a look at this in the next chapter). Each icon must be of exactly the same size and it's best to stick to multiples of 16 pixels for both the horizontal and vertical size of your icons. Note that when you save your bank, you must tell AMOS that the bank is to be saved as an 'Icon Bank' rather than a 'Sprite Bank'.

Once you've done that, you need to design each level using DPaint by picking up each block as a brush and then pasting it down to form a representation of the final display. DPaint is very good for this sort of work as it features a 'Grid' tool that allows you to lock all drawing operations to a grid of a fixed size. You can then take this representation and convert it into a series of data statements containing the numbers of the icons that are to be pasted down. To make this task somewhat easier, you may want to type little numbers into the blocks before drawing up the screen representation. Then all you have to do is to switch back and forth between AMOS and DPaint, reading off and then entering data statements as you go.



With all your data entered, all you need to draw the screen display is to load in the icon bank using the 'Load' command and then code a little loop that reads of each block number in turn and pastes it down onto the screen using the 'Paste Icon' command. The format of the 'Paste Icon' command is as follows:

---

Paste Icon X, Y, ICONNUM

 AMOS  
COMMAND DEFINITION

**X/Y** Not surprisingly, the 'X' and 'Y' parameters define the position on the screen where the icon is to be placed. Note that pastes the top left hand corner of the icon at these co-ordinates.

**ICONNUM** The 'IconNum' parameter is simply a value that tells AMOS which Icon in the 'Icon Bank' is to be pasted down.

```
Rem *** Screen Icons Demonstration
Rem *** By Jason Holborn
```



```
Screen Open 0,320,256,32,Lowres
Flash Off : Curs Off : Cls 0
```

```
Rem *** Load icons...
Load "AMOSBOOK:AbkFiles/DungeonBlocks.ABK"
Get Icon Palette
```

```
Rem *** Point read pointer to level data...
Restore LEVEL1
```

```
Rem *** Draw Screen from map data...
Screen Hide 0
For Y=0 To 15
  For X=0 To 19
    Read BLOCK
    Paste Icon X*16,Y*16,BLOCK
  Next X
Next Y
```



---

**Reserve Zone** `NUMBEROFZONES`


The 'Reserve Zone' command essentially tells AMOS to set aside an area of memory that will hold the co-ordinates of your screen zones. How much memory is set aside depends entirely upon how many screen zones that you tell it to reserve with the 'NUMBEROFZONES' parameter. Once you've done this, you can define each screen zone in turn using the following command:

---

**Set Zone** `ZONENUMBER, X1, Y1 To X2, Y2`


**ZONENUMBER** The 'ZoneNumber' parameter is a value between 1 and the number of zones that you've defined with the 'Reserve Zone' command. Note that the first zone is 1 and not 0.

**X1/Y1** The 'X1' and 'Y1' parameters define the top left hand corner of the screen zone. These co-ordinates should be entered as screen co-ordinates.

**X2/Y2** After setting the 'origin' of your zone with the 'X1' and 'Y1' parameters, you need to define its length and height by passing two values that define the bottom right hand corner of the zone. If, for example, you set the origin of the zone at (50,20), setting the 'X2' and 'Y2' parameters to (200,200) would create a screen zone that is 150 pixels across and 180 pixels down.

Once you've set up your screen zones, you can perform a check on them all to detect when the mouse pointer lies directly over one of your screen zones. The command to do this 'Mouse Zone' and it returns either a value of 0 (indicating that the mouse pointer isn't directly over a screen zone) or the number of the screen zone that the mouse pointer is over. If, for example, you created three zones numbered 1, 2 and 3 and the mouse pointer was moved over screen zone 2, the value returned by the 'Mouse Zone' command would be 2. Here's a quick demonstration listing.

```
Rem *** Screen Zones Demonstration
Rem *** Filename - ScreenZones.AMOS

Screen Open 0,320,256,8,Lowres
Flash Off : Curs Off : Cls 0

Load If "AMOSBOOK:Pictures/ZoneBackdrop.IFF"

Rem *** Reserve enough memory for 4 zones...
Reserve Zone 4

Rem *** Initialise zones...
Set Zone 1,18,17 To 103,50
Set Zone 2,21,134 To 134,194
Set Zone 3,191,37 To 292,142
Set Zone 4,229,174 To 288,235

Pen 3 : Paper 0

Do
  Rem *** Read zone under mouse pointer...
  A=Mouse Zone

  If A<>0
    Bell
  End If

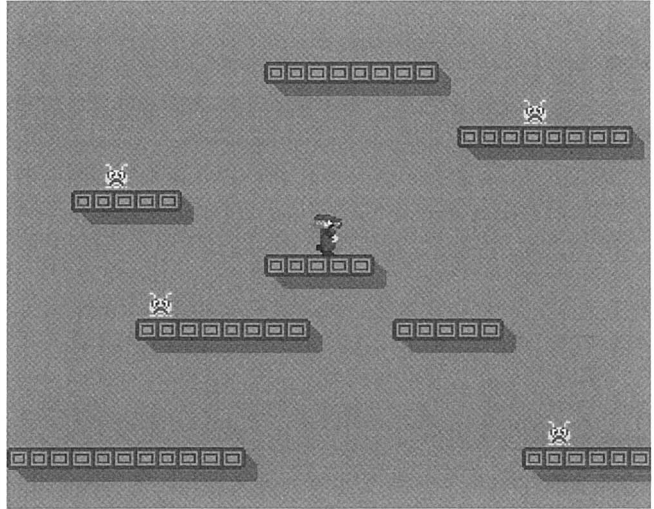
  Locate 2,11
  Print "Screen Zone = ";A;" "

  Wait Vbl
Loop
```



Checking for the position of the mouse pointer over a screen zone isn't very useful if you're writing a game that uses either blitter objects or hardware sprites other than the mouse pointer. For this purpose, AMOS provides two extra commands that can be used to check whether a set of co-ordinates lies within a screen zone. Simply by passing the 'hot spot'

Screen zones allow your bobs and sprites to interact with the background.



of an object (read more about hot spots in the next chapter) of an object to the appropriate function, you can check whether the object has moved over a screen zone. Unlike true collision detection, however, only a single point is checked, so it's quite possible that a large section of a bob could overlap into a screen zone before AMOS was aware of it. For a full demonstration of how to use screen zones in a game, check out the section on programming platform games at the end of this book.

Anyway, back to the two functions. AMOS provides two functions for checking when a set of co-ordinates lie within the boundaries of a screen zone – 'ZONE(X,Y)' and 'HZONE(X,Y)'. The only difference between these two functions is the co-ordinates systems that they handle – 'Zone()' works with screen co-ordinates and 'Hzone()' works with hardware co-ordinates (more on both of these co-ordinate systems can be found in the next chapter). For those of you that can't wait that long, here's a wonderfully sculptured listing!

```
Rem *** Bob Screen Zones Demonstration
```

```
Rem *** By Jason Holborn
```

```
Screen Open 0,320,256,8,Lowres
```

```
Flash Off : Curs Off : Cls 0
```



```
Load If "AMOSBOOK:Pictures/ZoneBackdrop.IFF"
Load "AMOSBOOK:AbkFiles/Helicopter.ABK"
Get Sprite Palette

SPRX=120 : SPRY=80
Hot Spot 1,$11
Bob 1,SPRX,SPRY,1

Rem *** Reserve enough memory for 4 zones...
Reserve Zone 4

Rem *** Initialise zones...
Set Zone 1,18,17 To 103,50
Set Zone 2,21,134 To 134,194
Set Zone 3,191,37 To 292,142
Set Zone 4,229,174 To 288,235

Pen 5 : Paper 0

Do
  If Joy(1) and 1
    SPRY=SPRY-2
  End If
  If Joy(1) and 2
    SPRY=SPRY+2
  End If
  If Joy(1) and 4
    SPRX=SPRX-2
  End If
  If Joy(1) and 8
    SPRX=SPRX+2
  End If

  Bob 1,SPRX,SPRY,1

Rem *** Read zone under mouse pointer...
```

```

A=Zone (SPRX,SPRY)

If A<>0
    Bell
End If

Locate 2,11
Print "Screen Zone = ";A;" "

Wait Vbl
Loop

```

---

## Screen compaction

When you're writing games or demos in AMOS, it's often handy to be able to store all the graphics that the program needs as part of the program itself. This removes the need to load the graphics in from disk every time the program is run (slowing your program down), and stops any would-be hackers from messing around with or even stealing your graphics.

Say, for example, you were writing a game in AMOS. If you were to load in the game's title screen every time the player finished a game, this would become very tiresome for the player. Not only that, but if you've been slaving away in DPaint for hours to produce a title screen, the last thing that you want is to see it hacked about by some hacker with about as much artistic (and programming) ability as a lemming.

As always, though, AMOS comes to the rescue with its powerful 'banks' facility. Banks are really nothing more than areas of memory that can be used to hold pictures, sounds, music modules and even sprites and bobs. In the case of all these examples except pictures, AMOS automatically places these files into their own banks, so their use is pretty much transparent. What's more, some of these banks are permanent so when you save your program out to disk, any banks that they use will be saved too as part of the program code. AMOS can also place the contents of entire screens into a memory bank too, so you can have all your game's graphics on tap instantly without having to load them first.



*If you own either AMOS 1.35 or Easy AMOS, then you'll have a utility called 'IFF\_Compactor.AMOS' that will pack screens automatically.*



However, there's one small problem – graphics eat up large amounts of memory. Even a low resolution screen with just 16 colours would eat up 40K. OK, so maybe a single picture isn't going to cause any memory problems, but imagine what would happen if you wanted to store just ten of these little beauties – you'd lose well over 400K of valuable memory! Amigas may come equipped with a minimum of 2Mb these days, but you have to bear in mind that there are still Amiga users out there who have nothing more than 512K OK, so they're living in the dark ages, but why limit your game to users with high-powered Amigas?

Again, AMOS comes to the rescue – with its very clever screen compaction routines that can crunch any screen down to a fraction of its original size. What's more, crunched screens are automatically placed into a permanent memory bank, so they're saved as part of your program. All you then have to do is to call AMOS's decompaction routines to view your crunched pictures in all their pixelised glory!

## **Screen packing**

So how do you go about packing a screen in the first place? Well, the first thing you need to do is to give AMOS something to pack by creating a screen with the 'Screen Open' command covered in chapter 4 and then either load a picture into it (using the 'Load IFF' command) or draw whatever you want into it using AMOS's vast array of drawing commands. Once you've got a screen, you can pack it. Under AMOS 1.35 and Easy AMOS, up to 16 memory banks can be used, not all of



which are available for holding packed pictures (if your game uses sprites or bobs, for example, AMOS will automatically steal a bank to hold these!). Under AMOS Professional, however, there's virtually no limit to the number of memory banks that you can use (well, you are limited to 65,536 actually, but I challenge you to use them all). The command to pack a screen into a memory bank is 'Spack':

---

**Spack** SCREENNUMBER To BANKNUMBER



**SCREENNUMBER** The Screennumber parameter is simply a pointer to a currently open screen.

**BANKNUMBER** Banknumber is a number between 0 and 16 (for Easy AMOS and AMOS 1.35 owners) or 0 and 65536 (for AMOS Pro owners) that identifies the bank that the screen is to be packed into.

It's best not to add this command into your program, simply because it only needs to be used once. Once a picture has been packed into a memory bank, you no longer need to either load the picture or call the 'Spack' command. I personally tend to halt a program once the picture I wish to pack has been displayed and then I pack the picture into a memory bank from AMOS's 'Direct Mode'. Packing a picture can take time, however, especially if the picture uses a lot of different colours and its high resolution. Once you have packed a picture, you can check to make sure that all went well by entering 'List Bank' in Direct mode. You should see something like this on your screen.

```
1 - Pac.Pic. S: $003FBAA8 L: 37192
```

This is AMOS's way of telling us that a packed picture (hence the 'Pac.Pic.' bit) is stored in memory bank 1. The 'S:' and 'L:' extensions tell you where in memory the picture can be found (just ignore this unless you're an advanced programmer) and the total size of the bank in bytes (in this case, the bank is approximately 37K in size). The 'L:' value doesn't always tell you a bank's size, however – if you load in either bobs, sprites or icons, this value will tell you how many bobs, sprites or icons are in the bank. Rather confusing I know, but bear this in mind.

Once you've got a picture safely packed into a memory bank, it will stay there until you remove it using the 'Erase BANKNUMBER' command. Even if you turn off your machine and then load your program back in again, it will still be there simply because AMOS saves all its permanent memory banks as part of your program listing. Anyway, to get it back, all you have to do is to open the screen which you'd like it decompressed into using the 'Screen Open' command and use the following command:

---

Unpack BANKNUMBER To SCREENNUMBER



**BANKNUMBER** The number of the bank that holds your packed picture.

**SCREENNUMBER** The number of the screen that you'd like your packed picture placed into. Note that when a picture is unpacked into a screen, all the settings of the original picture are restored too. So if you set up the original picture with a strange-sized viewport, this will be restored too.

Anyway, enough of the theory – why not try typing in this listing for an example of the power of these screen compaction routines:

```
Rem *** Screen compaction demo
Rem *** By Jason Holborn
```



```
FILENAME$=Fsel$("","","Please select IFF picture to load","","")
Load Iff FILENAME$,0
```

```
Screen Open 1,640,48,2,Hires
Screen Display 1,,240,,
Palette $60,$FFF : Cls 0 : Curs Off
```

```
Print "Picture loaded.. now let's pack it"
Print "Press left mouse button to start.."
```

```
_WAITMOUSE
```

```
Cls 0 : Print "Packing picture...."
Spack 0 To 1

Print "OK.. picture packed into memory bank 1"
Print : List Bank
Print : Print "Now let's unpack it into a couple of screen... ";
Print "Press left mouse button..."

_WAITMOUSE

For C=2 To 7
    Unpack 1 To C
    Screen Display C,,50+(C*20),,
Next C

Screen To Front 1
Screen 1 : Cls 0
Print "Eh voila! Five more screens unpacked from bank 1!"
Print "Press left mouse button to quit..."

_WAITMOUSE

For C=0 To 7
    Screen Close C
Next C

Erase 1 : Rem *** Erase packed picture bank
End

Procedure _WAITMOUSE
    Repeat
        Wait Vbl
    Until Mouse Key
End Proc
```

# Sprites and bobs

- Hardware and software 'sprites' (sprites and 'bobs')
- Creating sprites and bobs (objects)
- Displaying and moving objects
- 'Virtual' sprites
- Animating objects
- Flipping objects — 'hot spots'
- Collision detection
- Object Editor & menus
- The Animation Editor

If you're writing an arcade game or even a simple demo, chances are that you'll want to have sprites flying around the screen at high speed. I'm sure that even if you've only ever played a single arcade game in your life, chances are that game used sprites in one form or another. Virtually every arcade game, be it a shoot 'em up, a platform game or even 'weird' games like 'PaperBoy' and 'First Samurai', use sprites – a shoot 'em up, for example, will use a sprite for the player's spaceship, even more sprites for the alien nasties that attack you and all the game's missiles and other destructive paraphernalia will be sprites too. As you can see, sprites are a very important aspect of just about any arcade game!



**Sprites/  
'objects'**

Amiga sprites are somewhat confusing, however. Unlike lesser machines, the Amiga offers two different types of sprite – 'hardware' sprites and 'software' sprites (better known as 'Blitter Objects' or just plain 'Bobs'). The word 'sprite' should therefore be treated as a generic term that refers to both hardware sprites and blitter objects. Just to make life somewhat less confusing, however, I'll be using the term 'Object' whenever I mean to refer to both hardware sprites and bobs. Let's start then by taking a look at the strengths and weaknesses of each in turn.

## Hardware Sprites

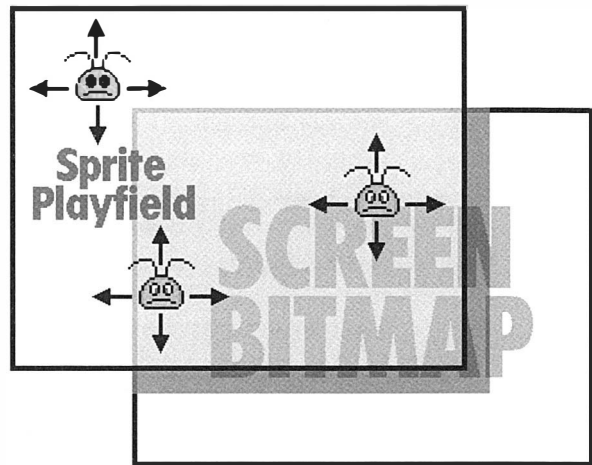


**Hardware  
sprites**

Hardware sprites are generated completely separately from a screen's bitmap by the Amiga's display chip Denise (or 'Lisa' on an AGA machine) on what is known as the 'sprite playfield', a sort of transparent sheet that is placed in front of all screens. As a result, they're almost completely independent of the screen they are being displayed on. All hardware sprites are low resolution only, so even if you open an AMOS screen in high resolution mode (640 by 512 pixels), any hardware sprites that you create will still be displayed in low resolution format. A good example of a hardware sprite is the mouse pointer that you use on the Amiga's Workbench to select icons and pull down menu items.

Hardware sprites are very fast indeed simply because they do not interact with a screen's bitmap whatsoever. So – unlike software sprites – the Amiga doesn't have to worry about redrawing any sections of the screen that a hardware sprite passes over. Hardware sprites do have their limitations, however. For starters, the Amiga's sprite generating

Hardware sprites are generated completely separate from the screen bitmap, on what the techies call the 'sprite playfield'.



hardware can only handle a maximum of eight sprites at any one time. Even then, you're restricted to just 3 colours per hardware sprite and each sprite can be no bigger than a maximum of 16 pixels across and 255 pixels down. If you try to display a sprite wider than 16 pixels, AMOS will join several hardware sprites together – a 48 pixel wide hardware sprite, for example, would be made up of 3 separate sprites, leaving just 5 spare. You could theoretically create a hardware sprite 128 pixels across, but that would use up all 8 hardware sprites in one go!



### Sprite size

It is possible to create 15-colour sprites but, because the Amiga's hardware simply layers two 3-colour hardware sprites (one on top of the other), using 15 colour hardware sprites immediately halves the total number of hardware sprites available from 8 to just 4.

Hardware sprites take their colour information from the screen that they are currently being displayed over. Each sprite reads its three colours from colour registers 17 through to 31 – hardware sprites 0 and 1, for example, read their colours from colour registers 17, 18 and 19. Sprites 2 and 3 read their colour information from registers 21, 22 and 23, Sprites 4 and 5 from registers 25, 26 and 27 and finally Sprites 6 and 7 read their colour information from registers 29, 30 and 31. Confusing I know, but that's the Amiga's display hardware for you!

AMOS does allow you to get around this limitation a little by making use of what are called ‘computed’ sprites. This clever bit of AMOS trickery essentially allows you to ‘re-use’ the same hardware sprite at different vertical positions down the screen. Unfortunately, strange things can happen if you attempt to align two computed sprites generated by the same hardware sprite along the horizontal axis. We’ll be covering these in quite some depth later within this chapter, so I won’t say too much about them at the moment.

### Software Sprites (Bobs)



#### Bobs

Software sprites (Bobs) are drawn directly into a screen’s bitmap by the Amiga’s ultra fast blitter chip. As a result, the number of colours that a bob can contain is limited only by the number of colours available on the screen that it is being drawn into, so you can create bobs with up to 64 colours (Extra Half Brite) if you so wish. AMOS automatically handles the process of drawing a bob into a screen and then restoring the screen’s original contents when the bob is moved, so they’re very easy to use indeed. Bobs have the added advantage of not being restricted in size either, so you can create truly massive bobs with ease.

The only real disadvantage of bobs is the speed at which the Amiga’s blitter chip can move them. Although the blitter is very, very fast indeed, bobs will never be as fast as true hardware sprites, especially if they are large and contain lots of colours. By restricting both the size and the number of colours used by a bob however, you’ll find that bobs are still the choice of most AMOS games programmers simply because they offer virtually unlimited flexibility with only a minor trade-off in speed. For games that require objects to move very fast however, hardware sprites may still be better.

---

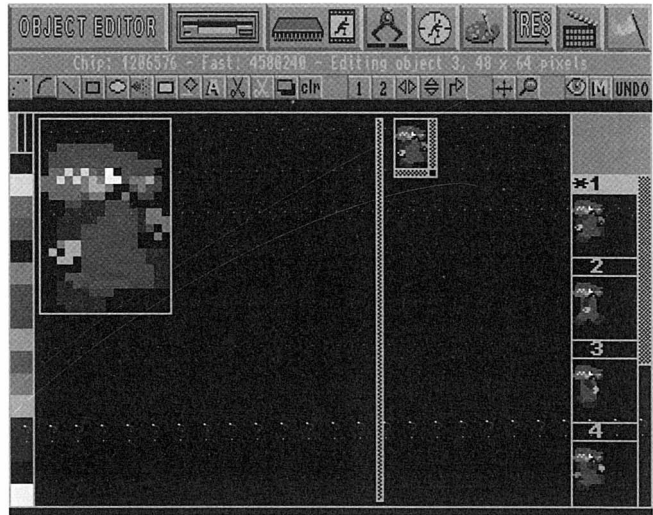
## Creating sprites and bobs



#### Sprite bank

In order to place either a sprite or a bob onto your Amiga’s screen, you must first define the appearance of the object that you wish to use by creating what is known in AMOS terms as a ‘Sprite Bank’. A Sprite Bank is simply a file that contains all the images that you want to attach to any sprites or bobs that you create. AMOS does allow you to ‘grab’ the image for an object directly from a screen using the ‘Get Bob’

Before you can display an object, you need to create a sprite bank using the Object Editor accessory bundled with your copy of AMOS.



command, but it's much simpler to define the image for all your sprites and bobs 'en masse' using the 'Object Editor' (or 'Sprite Editor', as it is called under AMOS 1.34 and Easy AMOS) accessory bundled with your AMOS interpreter.

I'm sure that I don't need to tell you how to use the Object Editor. If you're not entirely sure, it's more complex operations are fully documented at this end of this chapter.

Once you've created your Sprite Bank, you need to pull it into your AMOS program using the 'Load' command. Although this command is also used to load other types of banks (sample and music banks, for example), you don't have to worry about having to tell AMOS that you're trying to load a Sprite Bank – AMOS is intelligent enough to work that out for itself! If, for example, you had created a sprite bank called 'MySprites.Abk' (all bank files should end in '.Abk' so make them easy to recognise), all you'd have to do is to add the following line at the start of your program and AMOS will be able to load it into memory.



### Bank types

Load "DF0:MySprites.Abk"





If the colour palette of the screen that you intend to display your sprites and bobs on does exactly match the palette of the images in your sprite bank, the next step is to tell AMOS to transfer the sprite banks palette to the current screen using the following command:

### Get Sprite Palette



It's worth noting that if you've opened a screen that uses no more than sixteen colours and you want to display hardware sprites, you should still call this command even if the screen already has an identical palette. Even though the screen will not be able to take advantage of the extra colours used by the hardware sprites, it will still retain them, therefore allowing your sprites to be displayed in their true colours. Now, after all this jiggery pokery, we're ready to start using sprites and bobs...

## Displaying an object

Placing sprites and bobs onto the screen through any other programming language is a tiresome and somewhat long-winded process (especially if you're working with blitter objects), but AMOS makes it very simple indeed. All you need are two commands, one for each type of object. These two commands effectively turn on a sprite or bob and place it on the screen at the required position.

Sprite **NUMBER, X, Y, IMAGE**

Bob **NUMBER, X, Y, IMAGE**



**NUMBER** The 'Number' parameter is simply a value that is used by AMOS as an 'identifier' that allows it to keep track of each and every object on the screen. Every object must have its own unique identifier, although you can use the same number with two separate objects providing that they are of a different type (one's a sprite and the other is a bob). If you want to create a hardware sprite, then you're restricted to values between 0 and 7. Sprite number 0 is normally the Amiga's mouse pointer, so this cannot be used unless you turn it off (or 'free' it, as the techies would say) using the AMOS 'Hide' command.



**Efficient sprite  
use**

It's also worth noting that if you're attempting to display a 15-colour hardware sprite, not only does the maximum number of hardware sprites available drop from eight to four (don't forget that 15-colour sprites are effectively nothing more than two 3-colour sprites combined), but the 'Number' parameter must be even (0, 2, 4 or 6). If you specify an odd number, AMOS will be forced to use the next pair of available sprites, therefore wasting a sprite in the process. AMOS is thankfully much more generous with the number of blitter objects that you can create. By default, however, AMOS restricts itself to just 64 blitter objects, although this can be increased to a maximum of 255, providing you have enough memory. Don't forget that blitter objects not only eat up a lot of memory, but the more you use, the slower the screen is updated. On the whole, most games can get by with a maximum of 16 bobs on the screen at any one time.

**X/Y** Not surprisingly, the 'X' and 'Y' parameters define the position of the sprite or bob on screen. You should, however, be aware that although the 'Bob' command will happily accept co-ordinates in the form of screen pixel positions relative to the top left hand corner of the screen, hardware sprites use a slightly different system that is wholly incompatible with the pixel-based co-ordinates system used by AMOS's Bob and drawing commands.

Sprites use 'hardware co-ordinates' – a special co-ordinates system used by the Amiga's video hardware. Thankfully AMOS does provide us with a set of functions that allow us to convert to and from hardware and pixel-based co-ordinates in the form of the following functions:

---

```
PIXELXCOORD = X Hard(HARDXCOORD)
PIXELYCOORD = Y Hard(HARDYCOORD)
HARDXCOORD = X Screen(PIXELYCOORD)
HARDYCOORD = Y Screen(PIXELYCOORD)
```

---



Note how there are two functions for each type of co-ordinates system. This is necessary because a set of hardware X and Y co-ordinates with the same value (32, for example) would produce two different pixel-based X and Y co-ordinates. In this particular case, if both the X and Y

hardware co-ordinates were 32, the equivalent pixel-based X and Y co-ordinates would be 160 and 74 respectively. Confusing maybe, but the four functions above will make your life considerably easier.

**IMAGE** The ‘Image’ parameter is a number that tells the Sprite or Bob command which image in the sprite bank it should attach to this particular object. It’s worth noting that both sprites and bobs share the same sprite bank, so it is theoretically possible to assign the same image to both a sprite and a bob. Unlike the ‘Number’ parameter, you don’t have to give each object a unique image, so it’s perfectly possible to share the same image within the sprite bank amongst several objects.

---

## Moving an object

Moving an object around the screen is very easy indeed and involves nothing more than increasing or decreasing the ‘X’ and ‘Y’ co-ordinates of the object that you have created and then passing the new set of co-ordinates to the ‘Sprite’ or ‘Bob’ command. Increasing the value of an object’s X or Y co-ordinate will make it move to the left or down respectively and – not surprisingly – decrease either of these co-ordinates will have the opposite effect.

In order to produce smooth object movement, it’s best to store the ‘X’ and ‘Y’ co-ordinates of an object in a set of variables (‘MYBOBX’ and ‘MYBOBY’, for example) and then simply subtract or add values to these two variables. Each time you update these two variables, simply call the ‘Sprite’ or ‘Bob’ command to make the movement take effect.

Every time you move an object (or several objects), you should always wait for a vertical blank (using the ‘Wait Vbl’ command) before attempting to move the object (or objects) again. If you attempt to move the same object more than once during a single vertical blank, not only will your objects shoot around the screen so fast that you won’t be able to see them, but they will flicker terribly. If you need an object to move very fast, simply increase the amount that you increase or decrease the object’s co-ordinates by (adding a value of 4 to a co-ordinate will make an object move twice as fast than it would if you passed a value of 2).



**Screen  
updating**

Time for a quick demonstration. The following listing shows not only how to move both a hardware sprite and a bob across the screen, but also how to convert the set of pixel-based co-ordinates (called 'MYSPRX' and 'MYSPRY') used to hold the hardware sprite's X and Y position to the hardware co-ordinates required by the 'Sprite' command. When you run this demonstration, keep an eye on how the hardware sprite glides over the top of the blitter object and how certain areas of the sprite are transparent, allowing the blitter object to show through!



```
Rem *** Creating and Moving a Sprite Demonstration
Rem *** Filename - MoveObject.AMOS
```

```
Screen Open 0,320,256,4,Lowres
Flash Off : Curs Off : Cls 0
```

```
Load "AMOSBOOK:AbkFiles/Faces.ABK"
Get Sprite Palette
```

```
Double Buffer
```

```
Rem *** Free up mouse pointer!
Hide
```

```
Rem *** Define X and Y position of both objects
MYSPRX=0 : MYSPRY=128
MYBOBX=0 : MYBOBY=128
```

```
Do
  Sprite 0,X Hard(MYSPRX),Y Hard(MYSPRY),2
  Bob 0,MYBOBX,MYBOBY,1
```

```
  MYSPRX=MYSPRX+4
  MYBOBX=MYBOBX+2
```

```
  If MYSPRX>320 or MYSPRX<0
    MYSPRX=320-MYSPRX
  End If
```

```
If MYBOBX>320 or MYBOBX<0
    MYBOBX=320-MYBOBX
End If
```

```
Wait Vbl
```

```
Loop
```

---

## 'Virtual' sprites

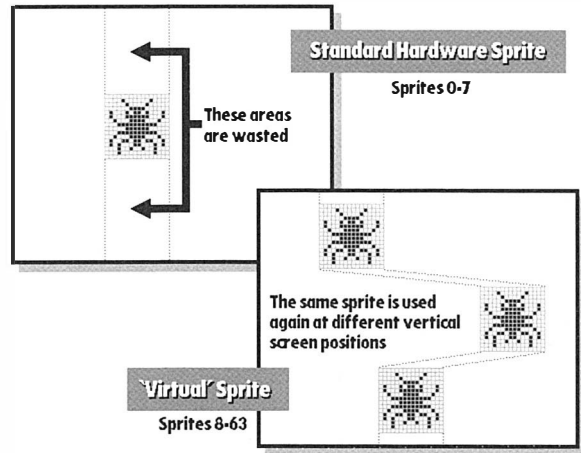
Hardware sprites are very fast and quite flexible in their own way, but they do have one big disadvantage that makes them somewhat less useful for most arcade games – you can only display eight of them at any one time and even then you're restricted to just three colours per sprite and each sprite can only be a maximum of 16 low resolution pixels across. AMOS does allow you to bear this limitation, however, thanks to a very clever bit of hardware trickery called 'virtual' (or 'computed') sprites.

Virtual sprites have a number of distinct advantages over normal hardware sprites. For starters, the number of virtual sprites that you can display on screen at once is considerably greater than the 8 object limit imposed by the Amiga's own hardware sprites. If they're used correctly, you can theoretically cram up to 56 virtual sprites on screen at once, each of which can be up to 128 or 64 low resolution pixels in width with either 3 or 15 colours. Don't get too excited though – virtual sprites do have their disadvantages. To understand these disadvantages, let's take a look at how AMOS generates virtual sprites.

Virtual sprites are based around the theory that although every normal hardware sprite can be up to 270 pixels in height, most of the sprite is effectively wasted every time you create a sprite smaller than this. However, thanks to the wonders of the Amiga's 'copper' co-processor (the chip responsible for the fantastic 'rainbow' effects we covered in the last chapter), the wasted sprite area can be 'reused' for another hardware sprite further down the screen. This system of 're-using' sprites is so powerful that a single hardware sprite can be split into up to 16 different 'virtual' sprites. In theory, when you display 16 virtual sprites, as far as the Amiga is concerned, it's only displaying a single hardware sprite.



*Virtual sprites allow you to increase the number of sprites that you can display simultaneously by 're-using' the same hardware sprite over and over again at different vertical screen positions.*



The only real problem with virtual sprites is that every virtual sprite generated by the same hardware sprite must not be displayed on the same set of horizontal scan lines (in theory, at least one scan line must be kept between each virtual sprite generated from the same hardware sprite). If, for example, you placed two virtual sprites at screen positions 20,30 and 20,60 they would be displayed fine. However, if you moved the second virtual sprite up so that it was alongside the first, the second sprite would disappear from view.

Virtual sprites are created and moved using exactly the same commands as hardware sprites, but in order to create a virtual sprite, you must specify a sprite number between 8 and 63 (sprites 0 to 7 are hardware sprites). Despite their limitations, however, virtual sprites are great for 'Galaxian' and 'Space Invaders'-style games.

Unfortunately, because of the amount of time required to keep track of a large number of objects, virtual sprites aren't quite as fast as their hardware-based counterparts simply because the code required to individually handle more than twenty or so virtual sprites will slow your program down. AMOS does compensate a little thanks to the 'Sprite Update Off' and 'Sprite Update' commands that allow you to turn off automatic redrawing of sprites and then manually draw them all yourself when you want. The 'Sprite Update Off' command should be placed at

the start of your program and then when you want to redraw all your sprites, simply call the ‘Sprite Update’ command. Even this technique won’t stop virtual sprites from appearing to move rather jerkily – if you want to rid your sprites of jerky movement altogether, you’ll need to compile your program. Try this listing for a demonstration of virtual sprites in action.

```

Rem *** Virtual Sprites Demonstration
Rem *** Filename - VirtualSprites.AMOS

FRAME=1 : FRAMEDELAY=0

NUMROWS=7 : Rem *** Number of rows of sprites
SPEED=2 : Rem *** Speed of sprite movement
POSITION=0 : Rem *** Number of movement steps counter
DROPOFFSET=0 : Rem *** Y offset of all sprites
DROPRATE=10 : Rem *** How far the sprites drop down the screen

Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0

Load "AMOSBOOK:ABKFiles/SpaceInvader.ABK"
Get Sprite Palette
Sprite Update Off

Rem *** Set up colour palette of all sprites
For C=0 To 3
    Colour 17+(C*4), $F00
    Colour 18+(C*4), $F
    Colour 19+(C*4), $FF0
Next C

Rem *** Turn off mouse pointer!
Hide

Rem *** Initialise Positions of 56 virtual sprites
Dim INVADER(NUMROWS*8,2)

```



```
INVADERNUM=0
For B=0 To NUMROWS-1
  For C=0 To 7
    INVADER(INVADERNUM,0)=C*25
    INVADER(INVADERNUM,1)=B*20
    INVADERNUM=INVADERNUM+1
  Next C
Next B

Repeat
  Rem *** Move all virtual sprites
  For ROW=0 To NUMROWS-1
    OFFSET=ROW*8
    For C=0 To 7
      INVADER(C+OFFSET,0)=INVADER(C+OFFSET,0)+SPEED
      X=X Hard(INVADER(C+OFFSET,0))
      Y=Y Hard(INVADER(C+OFFSET,1)+DROPOFFSET)
      Sprite C+OFFSET+8,X,Y,FRAME
    Next C
  Next ROW

  Rem *** Reverse direction of aliens if they
  Rem *** reach edge of screen

  POSITION=POSITION+SPEED
  If POSITION>130 or POSITION<-130
    SPEED=-SPEED
    POSITION=0
    DROPOFFSET=DROPOFFSET+DROPRATE
    If DROPOFFSET>114
      DROPOFFSET=0
    End If
  End If

  Rem *** Update animation frame number

  If FRAMEDELAY=5
```



```
FRAME=FRAME+1
If FRAME=3
    FRAME=1
End If
FRAMEDELAY=0
End If
FRAMEDELAY=FRAMEDELAY+1

Rem *** Draw all virtual sprites now
Sprite Update

Rem *** Wait for vertical blank before proceeding
Wait Vbl
Until Inkey$<>""
```

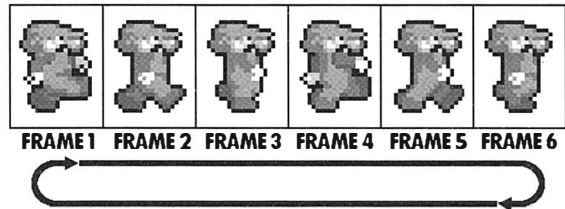
---

## Object animation

Virtually all games feature some form of object animation, where the central sprite (the sprite controlled by the player) and any other objects on the screen are animated to add a greater feeling of life and movement to them. Take a game like DMA Design's 'Walker' – not only does the Walker sprite literally 'walk' back and forth across the screen, but its cannons and missile launchers realistically recoil whenever the player fires the walker's cannons. Without animation, the Walker sprite would look rather silly as it simply glided across the play area. Even a basic game can benefit from animation, even if the animation is fairly simple.

Animating an object is very simple indeed. All you need to do is to draw up the 'frames' that will form the animation within the AMOS object editor as a series of separate object images running in sequence from the first frame to the last frame. Designing complex animations such as a character running or walking is quite difficult unless you're an experienced animator, but writing the code to handle that animation is considerably easier. The simplest way of designing a complex animation such as a character walking is to take an existing animation (such as the 'Mario' sprite bank on the AMOS 'Sprites 600' disk – this is still available from various public domain libraries if you don't already own it) and simply modify the graphics to suit your needs.

Any object can be animated simply by changing the image that it displays at regular intervals. This 'Super Mario' object, for example, can be found on the 'Sprites 600' disk bundled with the original AMOS.



Taking all the frames that you've designed and animating an object with them is very simple too. Say, for example, you had defined a sprite bank containing an animation of a horse running. If the frames within the sprite bank were numbered from 1 to 10, all you'd have to do is to cycle through these frames in sequence, one frame every vertical blank. The current frame number is placed into a variable (something like 'CURRENTFRAME', for example) which is then passed to the 'IMAGE' parameter of the 'Sprite' or 'Bob' commands. You may well find that your animation runs too fast – after all, if the animation is updated every vertical blank, then it will run at a rate of 50 frames per second on a PAL Amiga or 60 frames per second on an NTSC Amiga.



### Slowing down animations

The best way of slowing down an animation is to include some sort of delay counter that is increased every vertical blank. Then, when the delay counter reaches a certain value, you update the animation and reset the delay counter and start counting again. If, for example, you only updated the animation everytime the delay counter reached 5, the animation would be updated at a rate of 10 frames per second ( $5 \times 10 = 50$  vertical blanks = 1 second).

Objects can also be animated very easily using AMOS's very powerful 'AMAL' animation language which we'll be covering in a later chapter. In the meantime, however, try this very simple listing for size:

```
Rem *** Object Animation Demonstration
Rem *** Filename - ObjectAnimation.AMOS
```



```
Screen Open 0,320,256,32,Lowres
```

```
Flash Off : Curs Off : Cls 0

FRAME=1 : Rem *** Current frame number
DELAY=0 : Rem *** Animation delay counter
SPEED=5 : Rem *** Speed of animation in vertical blanks

Rem *** Try changing the 'SPEED' variable to increase or
Rem *** decrease the speed of the animation

Load "AMOSBOOK:ABKfiles/Mario.ABK"
Get Sprite Palette

Do
  Rem *** Is it time to update animation yet?
  If DELAY=SPEED

    Rem *** Increase frame number
    FRAME=FRAME+1

    Rem *** Loop back around if all frames displayed
    If FRAME=7
      FRAME=1
    End If

    Rem *** Rest delay variable
    DELAY=0
  End If

  Rem *** Add 1 to delay
  DELAY=DELAY+1

  Rem *** Draw Bob
  Bob 1,160,128,FRAME

  Wait Vbl
Loop
```

---

## Interrupt-driven animations

If you're creating a particularly complex object animation, you may find it rather difficult and certainly code-intensive to update the animation manually. Although manual control of an animation gives you far greater control over the animation process, you may find it easier just to let AMOS handle the task for you. AMOS provides a very handy 'Anim' instruction that runs under interrupt, so your program is free to do its stuff without you having to worry about updating the animation yourself. What's more, the 'Anim' command also includes a very handy 'delay' parameter which removes any need for you to slow down an animation yourself.

The 'Anim' command runs under AMOS's powerful AMAL animation language, which we shall be taking a look at in chapter 9. The format of the 'Anim' command is as follows:

---

**Anim** CHANNEL,“(IMAGE, DELAY), (IMAGE, DELAY)...L”



**CHANNEL** The 'Channel' parameter is a number between 1 and 16 that tells AMOS which of its interrupt channels the animation is to be assigned to. These interrupt channels are exactly the same as those used by AMOS's animation language AMAL, so it's not possible to assign both an animation and an AMAL program to the same channel. Similarly, each animation must be assigned to its own unique channel.

**IMAGE** Each and every frame that you wish to be included in the animation must be enclosed in a set of brackets complete with two parameters. There's virtually no limit to the number of frames that you can assign to an animation. The first of the two parameters required by each frame is the 'image' parameter which, not surprisingly, tells AMOS which object image in the Sprite Bank it should display for a particular frame. What's more, the same image can be used over and over again at different positions within an animation, so some quite complex animations can be created with ease.

**DELAY** The ‘Delay’ parameter tells AMOS how long a particular frame is to be displayed before moving on to the next. Each delay unit lasts exactly 1/50th of a second (one vertical blank) and you’re free to specify any size of delay. For example, a delay value of ‘100’ would make AMOS wait 2 seconds before updating the animation again ( $100 = 2 \times 50 = 2$  seconds).

**‘L’** Normally AMOS will run through an animation and then stop when it reaches the last frame, but you can force it to loop your animation by putting a capital ‘L’ after the last set of frame parameters.

### Running an animation

In order to get your animation to run, a few extra operations must be performed in the following order.

- 1 First, the object (be it a sprite or a bob) should be created and placed on the screen using either the ‘Bob’ or ‘Sprite’ commands covered earlier.
- 2 Next, the interrupt channel that is to handle the animation should be assigned to the object you wish to animate using the ‘Channel’ command. For the purpose of animating an object, the ‘Channel’ command has two forms:

---

Channel CHANNELNUMBER To Bob BOBNUMBER

Channel CHANNELNUMBER To Sprite SPRITENUMBER



The ‘ChannelNumber’ parameter is simply a value between 1 and 16 which tells AMOS which of its sixteen interrupt channels are to be used to run your animation. Secondly, both forms of the ‘Channel’ command expect to be fed the number of the sprite or bob that you wish to animate. This should be exactly the same identifier number that you specified when the object was first created.

- 3 With the interrupt channel assigned, you can then define your animation string using the ‘Anim’ command covered above. Note that in order to get the interrupt channel that you defined in step 2 to recognise your animation, you must specify the same channel number.

- 4 Finally, the animation is ready to run and all that is needed is the ‘Anim On’ command. This turns on the interrupt channel and sets the animation in motion.

### Changing the sequence of frames

Once you’ve created an animation, you may want to change it so that a different sequence of frames are displayed. This can be particularly handy if you’re writing an arcade game that requires an object to act different according to what it is doing – in a beat ‘em up game, for example, you could easily assign an animation to each martial arts move the character is capable of performing. Thankfully, AMOS allows you to do this too without having to go through the hassle of setting the entire animation up from scratch. All you need to do is to turn off the animation using the ‘Anim Off’ command, define your new animation with the ‘Anim’ command and then simply turn the animation back on again with the ‘Anim On’ command.

It’s worth noting, however, that each time you change an object’s animation, the same channel number must be specified for each new animation string in order for the interrupt channel that you have assigned to the object to recognise the new animations. What could be simpler? And, just to prove it, here’s a short demonstration listing:

```
Rem *** Anim Command Demonstration
Rem *** By Jason Holborn

Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0

Load "AMOSBOOK:AbkFiles/Mario.ABK"
Get Sprite Palette

Bob 10,145,128,1
Channel 1 To Bob 10

Anim 1,"(1,4)(2,4)(3,4)(4,4)(5,4)(6,4)L"
Anim On
```



```
Pen 1 : Paper 0
Locate 0,5 : Centre "Press any key to change animation"
Wait Key
```

```
Anim Off
Anim 1, "(6,10) (2,4) (6,10) (2,4)L"
Anim On
```

```
Locate 0,7 : Centre "Press any key to quit"
Wait Key
```

---

## Object 'flipping'

Unless your objects all move in a single direction, it's often handy to be able to adjust their orientation so that they face in the direction that they are moving. The most obvious way of achieving this would simply be to design a set of images for each direction of movement and then change to another set whenever the object changed direction. Although you can do this in AMOS, it is rather wasteful of memory because you'll need four sets of object images (one for each direction) instead of one.

AMOS comes to the rescue by letting display the same set of images in different orientations. All you need are these three very basic functions:

---

```
FLIPPEDIMAGE = Hrev(IMAGE)
```



which flips an image in the sprite bank identified by the parameter 'IMAGE' horizontally,

---

```
FLIPPEDIMAGE = Vrev(IMAGE)
```



which flips an image vertically, and

---

```
FLIPPEDIMAGE = Rev(IMAGE)
```



which flips an image both horizontally and vertically. In all three cases, the image number of the flipped image is stored into the variable

‘FLIPPEDIMAGE’ (this is only an example, so you can call this variable whatever you like). It’s worth noting that these three flipping functions won’t flip an image back to its original orientation once they have been flipped. In order to do this, you must restore the object back to its original orientation by passing the original image number (‘1’, for example).

The value stored in the variable ‘FLIPPEDIMAGE’ has a very distinct format that can easily be manipulated. If you call the ‘Hrev()’ function, a value of hex \$8000 is simply added to the image number that you passed to the function (passing an image number of ‘4’, for example, would return a value of ‘\$8004’). As you can see, you can quite easily horizontally flip objects yourself just by adding hex \$8000 to the image number yourself. The other two functions (‘VRev()’ and ‘Rev()’) add hex \$4000 and \$C000 with the image number respectively. If you know your hexadecimal, you may notice that the value returned by the ‘Rev()’ function (\$C000) is simply the two values returned by the ‘Hrev()’ and ‘VRev()’ functions added together (\$4000 + \$8000 = \$C000).



To be perfectly honest, AMOS’s image flipping commands are a bit inflexible because they won’t automatically flip an image back to its original orientation if they are called a second time, which is somewhat annoying. I personally find it much easier to simply use the values that the functions add to the image number and then manually perform a ‘XOR’ logic operation on the image number. For example, a line such as

```
NEWIMAGE = IMAGE xor $8000
```



would give you a value of \$8004 if the value in ‘IMAGE’ was 4, flipping the image horizontally in the process. If you then called the same line again, \$8000 would be removed, effectively horizontally flipping the image in the opposite direction. Clever, eh?

It’s worth noting that AMOS cannot rotate an image in the sprite bank, so you’ll still need to create a second set of images for your objects if you want an object to appear to rotate 90 degrees. Anyway, here’s a quick demonstration of object flipping and hot spots (which we’ll be covering next) in action.





```
Rem *** Bob flipping demonstration
Rem *** Filename - ObjectFlipping.AMOS

Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0

Load "AMOSBOOK:Abkfiles/Hand.abk"
Get Sprite Palette

SPRX=160 : SPRY=128 : FRAME=1

Rem *** Set hot spot to middle of image
Hot Spot 1,$11

Do
  If Joy(1) and 4
    If SPRX>0
      SPRX=SPRX-2
    End If

    Rem *** Get Flipped Image
    FRAME=Hrev(FRAME)
  End If

  If Joy(1) and 8
    If SPRX<320
      SPRX=SPRX+2
    End If

    Rem *** Restore Image
    FRAME=$1
  End If

  Bob 1,SPRX,SPRY,FRAME
  Wait Vbl
Loop
```

**'Hot spots'**

Flipping an image correctly isn't just a case of calling the appropriate function, however. Whenever an object is placed onto the screen, it is positioned relative to an invisible point called the 'hot spot' which, by default, is placed at the top left hand corner of all sprites and bobs. If you're simply moving an object around the screen, the default hot spot setting is fine – but you may find it rather limiting when you come to flip an object either horizontally or vertically. The hot spot is also used to define the axis along which the object is flipped and therefore the default setting will cause an object to be 'mirrored' rather than flipped along the object's centre point. This can be rather annoying when you need to flip and object so that it appears to turn around – a spaceship, for example.

AMOS comes to the rescue here too with a handy little function called 'Hot Spot' that – not surprisingly – allows you to adjust the hot spot setting of any image in the sprite bank. Note that whenever you change the hot spot setting of an image, all objects that use that image will automatically have their hot spot setting adjusted accordingly. The format of the 'Hot Spot' command is as follows:

---

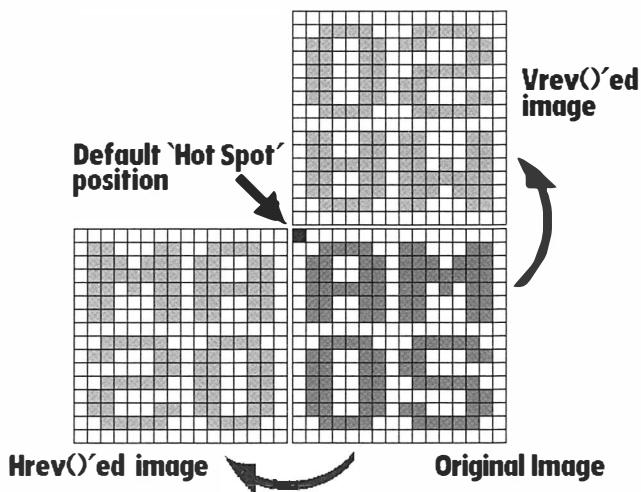
**Hot Spot IMAGE, X, Y**

**IMAGE** Not surprisingly, the 'Image' parameter is simply a value that contains the number of the image in the sprite bank that you wish to work on.

**X/Y** These two parameters define the position of the hot spot relative to the top left hand corner of the image (position 0,0). If, for example, you wanted to place the hot spot exactly 10 pixels across and 20 pixels down from the top left hand corner of the object, you'd simply pass a value of 10 for the 'X' parameter and 20 for the 'Y' parameter.

Obviously there's one big disadvantage with this approach – you're left to do the hard work of having to calculate exactly where you want the hot spot placed. Wouldn't it be so much easier if you could simply tell AMOS to place the hot spot in the middle of the object or even in the bottom right hand corner? Well folks, AMOS can do this too thanks to a second version of the 'Hot Spot' command:

Unless you adjust an object's 'Hot Spot', objects will not be flipped correctly. Instead, AMOS simply 'mirrors' the image, as this diagram shows.



## Hot Spot IMAGE, PRESET

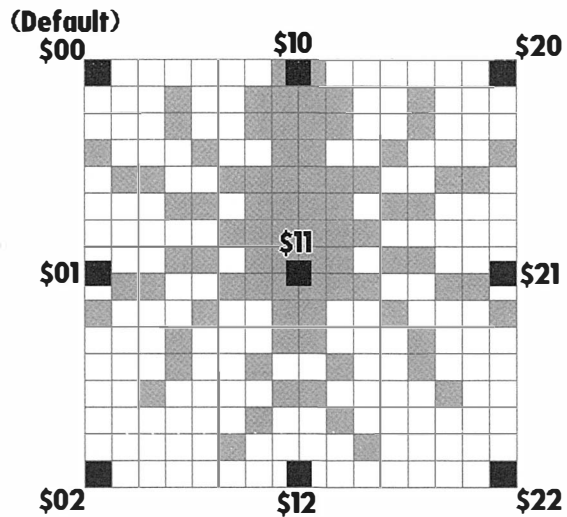


**IMAGE** The 'Image' parameter is exactly the same as the first version of the command. Simply tell it which image in the sprite bank you wish to work on.

**PRESET** Built into AMOS are a set of nine preset hot spot positions that cover the most commonly used hot spot settings. These settings are as follows:

- \$00** Top left hand corner of image (default position)
- \$01** Middle of left edge
- \$02** Bottom left hand corner
- \$10** Middle of top edge
- \$11** Middle of image
- \$12** Middle of bottom edge
- \$20** Top right hand corner
- \$21** Middle of right edge
- \$22** Bottom right hand corner

*Changing an object's 'hot spot' setting controls not only the object's 'handle', but also the axis setting when the object is flipped.*



## Collision detection

Having hordes of sprites and bobs whizzing around the screen is all very well if you're writing nothing more than a fancy demo, but they need to be able to interact if you're writing just about any form of arcade game. If you're writing a shoot 'em up game, for example, you need to be able to detect when the player's missiles strike an attacking alien spaceship or vice-versa. If you're unable to detect when an object comes into collision with another, all your objects are going to move past (and even through!) each other totally oblivious to the fact that they've come into contact with one another.

The technique of detecting when objects have collided is called – not surprisingly – 'collision detection' and it's used in just about every arcade game you could possibly imagine. If the game needs to be able to sense when two or more objects come into contact, then collision detection is used.

Not surprisingly, AMOS allows you to detect collisions too thanks to a powerful array of functions designed specifically for the task. Detecting collisions from within any other programming language is notoriously difficult, but AMOS makes it an absolute doddle.

In order to detect collisions, AMOS uses what is known as a ‘mask’ around all objects that you create. These masks aren’t just used for collision detection, however – because the Amiga’s blitter works on whole multiples of 2 bytes only (16 bits), the mask is used to ensure that any pixels around the image coloured in the background colour (colour 0) are made transparent, therefore allowing objects to be displayed in just about any irregular shape that you define (a spaceship, for example). By default, AMOS automatically creates a mask for all blitter objects, but it’s up to you to tell it to create a mask for an image whenever you intend to use it as a hardware sprite. If you don’t create a mask for a hardware sprite, AMOS will be unable to detect collisions between it and other objects. The command to create a mask for a hardware sprite image is as follows:

---

#### Make Mask IMAGE



**IMAGE** The ‘Image’ parameter is simply the number of the image in the sprite bank that you wish to create a mask for. For example, if you wanted to create a mask for image number 1 in the sprite bank, you’d pass a value of 1. This parameter is optional. If you want to create a mask for all the objects in the sprite bank, just enter the ‘Make Mask’ command on its own.

Once you’ve created a mask for all your sprite images, you’re ready to start checking for collisions. For this purpose, AMOS provides four different functions, each of which is designed to handle a collisions between different combinations of objects. They are as follows:

---

**STATUS = Bob Col (BOBNUMBER)**

**STATUS = Bob Col (BOBNUMBER, FIRSTBOB To LASTBOB)**



The first function checks for collisions between the Bob you specify with the ‘BOBNUMBER’ parameter and any other bobs. The second form of this function allows you to limit the number of bobs that the Bob you specify can collide with by passing a range of Bob numbers between the value held in ‘FIRSTBOB’ and the value held in ‘LASTBOB’). For

example, if you wanted to check whether the player's spaceship (Bob number 0) had collided with a series of alien Bobs numbered from 1 to 10, something like the following would do the job.

```
STATUS = Bob Col(0,1 To 10)
```



This can be handy if you're writing a game where only collisions with certain bobs will have an effect (missiles colliding with a spaceship, the player's bob running into a power-up etc).

```
STATUS = Sprite Col(SPRNUMBER)
```

```
STATUS = Sprite Col(SPRNUMBER, FIRSTSPR To LASTSPR)
```



The 'Sprite Col()' function detects collisions between the hardware sprite that you specify with the 'SPRNUMBER' parameter and any other hardware sprites on the screen. Just like the 'Bob Col()' function, the 'Sprite Col()' function can limit the range of sprites checked by limiting the collision detection to a specified range of hardware sprites.

```
STATUS = Spritebob Col(SPRNUMBER)
```

```
STATUS = Spritebob Col(SPRNUMBER, FIRSTBOB To LASTBOB)
```



Both of these first two types of collision detection work on only a single type of object – 'Bob Col()', for example, will only detect collisions between Bobs and 'Sprite Col()' only works with hardware sprites. The 'Spritebob Col()' function, however, lets you detect collisions between a specified hardware sprite (SPRNUMBER) and either all Bobs on the screen or a specified range of Bobs.

```
STATUS = Bobsprite Col(BOBNUMBER)
```

```
STATUS = Bobsprite Col(BOBNUMBER, FIRSTSPR To LASTSPR)
```



Finally, we have the 'Bobsprite Col()' function which, detects collisions between a single specified Bob (BOBNUMBER) and either all hardware sprites or a specified range of hardware sprites.

If any of these functions detect a collision, AMOS places a value of '-1' into the variable 'STATUS'. If no collisions are detected, a value of '0' is returned. Using the 'Bob Col()' function as an example, you could therefore detect whether a collision had taken place using something like the following snippet of code:

```
STATUS = Bob Col(0,1 To 10)
If STATUS = -1
    Print "Collision detected!"
Else
    Print "No collisions took place."
End If
```



Obviously there's one big problem with this – although AMOS tells you that a collision has taken place, it doesn't tell you which object collided with which. This is where the 'Col()' function comes in. The 'Col()' function allows you to find out which objects were involved in a collision. The format of the 'Col()' function is as follows:

```
STATUS = Col (OBJECTNUMBER)
```



**STATUS** The value returned by the 'Col()' function is placed into this variable. Obviously you can call it whatever you want, but I've used 'STATUS' just to keep things nice and neat. The return value will depend entirely upon whether the object specified with the 'OBJECTNUMBER' was involved in a collision. If it was, a value of '-1' is returned otherwise a value of '0' is returned.

**OBJECTNUMBER** The 'ObjectNumber' parameter is the number of the object that you'd like to check. If you tested for a collision between object '0' and objects '1' through to '10', for example, you would have to check objects '1' through to '10' individually to see if they had collided with object '0'.

Using the 'Col()' function is a bit like checking to see what cars in your street have dents in them when you wake up in the morning, only to find that your car has been hit during the night. In real life, this is hardly a

reliable way of nailing the culprit, but it's AMOS equivalent is very reliable because the status of each and every object is updated every time one of the four collision detection functions is called. So if a bob was involved in a collision before you called the 'Bob Col()' function, for example, it would have been sent off to the body shop, hammered out, resprayed and put back onto the road (in a manner of speaking at least!) by the time you call the 'Bob Col()' function again. Obviously if it's been involved in a collision again, then the scars will be there and the 'Col()' function will detect them.

One question does arise, however – what happens if you have a series of hardware sprites that use the same set of numbers as a series of Bobs (Sprite 0 and Bob 0, Sprite 1 and Bob 1 etc)? Well, the 'Col()' function isn't stupid – it knows automatically that if you've called a collision detection function that checks for collisions with sprites, then it will check sprites and if you're checking for collisions with Bobs, then it will check Bobs. As I said, AMOS isn't stupid.

So how do you go about detecting which objects collided with your sprite or Bob? Simple – you just tell the 'Col()' function which object you'd like it to check and it will either return a value of '-1' or '0'. If a value of '-1' is returned, then you know that the object in question collided with your sprite or bob. If you want to check a series of objects, the easiest way is to use a simple FOR...NEXT loop such as the one used in the following snippet of code:

```
STATUS = Bob Col(0,1 To 10)
If STATUS = -1
  For A = 1 To 10
    If Col(A) = -1
      Print "Object";A;" was involved in a collision!"
    End If
  Next A
End If
```



This short piece of code simply checks to see whether objects '1' through to '10' were involved in a collision. If they were, a short message is printed on the screen. Fancy a fully working demonstration?





```
Rem *** Collision Detection Demonstration
Rem *** Filename - CollisionDetection.AMOS

Screen Open 0,320,256,32,Lowres
Flash Off : Curs Off : Cls 0

Pen 1 : Paper 0
Locate 0,1
Centre "Knock heads together with the joystick!"
Double Buffer

Load "AMOSBOOK:AbkFiles/Faces.ABK"
Get Sprite Palette

Rem *** Create collision mask for hardware sprite
Make Mask 3
Rem *** Turn off mouse pointer to free up sprite
Hide

Rem *** Give sprite DMA chance to catch up
Wait 1

SPRX=160 : SPRY=128 : SPRNUM=0
Global SPRX,SPRY

Rem *** Create 4 Bobs and 2 Hardware Sprites

Bob 1,40,50,2
Bob 2,110,50,2
Bob 3,190,50,2
Bob 4,260,50,2

Sprite 0,X Hard(75),Y Hard(200),3
Sprite 4,X Hard(230),Y Hard(200),3

Do
    _MOVEFACE
```

```
Rem *** Has Bob hit another bob?
STATUS=Bob Col(SPRNUM)
If STATUS=-1
    Rem *** Which Bob did it hit?
    For C=1 To 4
        If Col(C)=-1
            Boom
            TXT$=" Collided with Bob number"+Str$(C)+" "
            Locate 0,14
            Centre TXT$
        End If
    Next C
End If

Rem *** Has Bob hit a hardware sprite?
STATUS=Bobsprite Col(SPRNUM)
If STATUS=-1
    Rem *** Which Sprite did it hit?
    For C=0 To 4 Step 4
        If Col(C)=-1
            Boom
            TXT$=" Collided with Sprite number"+Str$(C)+" "
            Locate 0,14
            Centre TXT$
        End If
    Next C
End If

Bob SPRNUM, SPRX, SPRY, 1
Wait Vbl

Loop

Procedure _MOVEFACE
    If Joy(1)=1 and SPRY>16
        SPRY=SPRY-4
    End If
```

```
If Joy(1)=2 and SPRY<230
    SPRY=SPRY+4
End If
If Joy(1)=4 and SPRX>16
    SPRX=SPRX-4
End If
If Joy(1)=8 and SPRX<300
    SPRX=SPRX+4
End If
End Proc
```

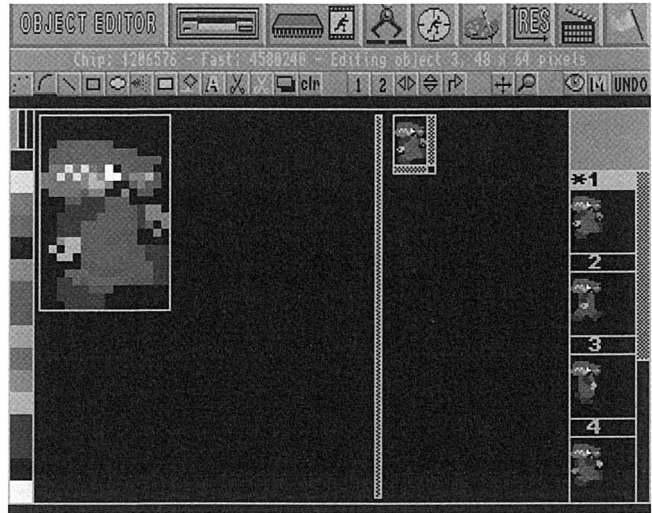
---

## The Object Editor

Before you can use sprites or bobs within your AMOS programs, you must first create a sprite bank using the powerful 'Object Editor' accessory bundled with AMOS. The Object Editor is essentially nothing more than a paint program geared towards the task of drawing images for use with AMOS Sprites and bobs. Just like a conventional paint package such as Electronic Arts' Deluxe Paint, the AMOS Object Editor provides you with a whole range of drawing tools including freehand, circle, box and line etc. What's more, it can even import images stored in standard IFF format, so there's no reason why you couldn't use your favourite paint package to design your object images and then simply use the Object Editor to pull them into a sprite bank. It can also be used to draw 'Icons' and 'Blocks', but we'll cover this aspect of the Object Editor later.

We don't really have the space to document the object editors bundled with all three different versions of AMOS, so we'll be concentrating on the Object Editor bundled with AMOS Professional. This is virtually identical to the Object Editor including with Easy AMOS, so Easy AMOS users will find the following information useful too. I'm sorry if you are using the original AMOS, but then isn't it time you upgraded anyway? With AMOS Professional now selling for peanuts, you're missing out big time if you don't upgrade. Not only that, but AMOS's Object Editor is so dire (sorry Aaron!), that it's worth buying AMOS Professional just for the vastly improved object editor.

*The AMOS Professional Object Editor is so good that it alone makes upgrading to AMOS Pro worthwhile.*



So how does the AMOS Professional Object Editor work? Well, as I said earlier, it's very similar to a conventional paint program. Of course there are differences – for starters, the object editor isn't restricted to just a single screen. Each object image that you create is allocated its own 'frame' that just happens to correspond with the image numbers required by AMOS's sprite and bob commands. These frames can be of any size, although the horizontal size is always a multiple of 16. This is necessary because both the Amiga's sprite hardware and blitter work on multiples of 16 too, so the Object Editor makes sure that your object image is always of the required size.

When you first boot up the Object Editor (either by selected 'Edit Objects' from AMOS Professional's 'User' menu or by selecting 'Bob Editor' from Easy AMOS' 'System Menu'), you'll be presented with a very pretty-looking main menu screen that contains a strip of icons along the top and the main object editing screen below this. The object editing screen is split into three areas – a very thin palette selector along the left hand side, the bank frame selector (allows you to select any image from the current sprite bank simply by double clicking on it) along the right and the rest of the display is split between two views of the object that you're currently editing (one is magnified and the other is normal size).

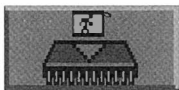
The real power of the Object Editor is hidden away behind that obscure line of icons that run along the top of the object editor screen just above the line of drawing tool icons. Most of them are fairly self-explanatory – the disk drive icon, for example, brings up the disk menu that allows you to load, merge, save and ‘save as’ (save under a different filename) a sprite bank. Some are not so straightforward, so let’s take a look at them in a bit more detail.

## The Bank Menu

The Bank menu gives you extensive control over the current sprite bank, letting you to extract and replace, insert and delete object frames easily.



**Get Object** Essentially the same as double-clicking on one of the images shown in the bank frame selector, the ‘Get Object’ icon will pull the image held in the current frame into the main editing screen, allowing you to work on it.



**Put Object** Every time you make a change to an image you’ve extracted from the sprite bank, you must put it back into the bank using this option. To be perfectly honest, you don’t have to use this option – if you select a different frame from the one you are currently editing, the Object Editor will automatically put it back for you.



**Put Object To** This option allows you to place the object you are currently editing into any of the frames available in the current sprite bank. If the frame that you select already holds an image, it will be replaced with the new image.



**Insert Object** The ‘Insert Option’ icon allows you to insert the image being edited directly into the sprite bank at the current position. If any icons proceed that position, they will be shunted down one position to make room.



**Delete Object** Lets you delete the currently selected frame from the sprite bank. If there are sprites after the current frame, the resulting gap will be closed effectively decreasing the image number of all frames that follow it.



**New Bank** Wow! Heavy stuff. If you click on this icon, the entire contents of the sprite bank that you are currently editing will be wiped from memory completely. Once you delete the sprite bank, there's no way of getting back unless you saved it to disk first.



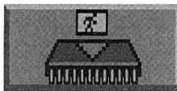
**Auto Get/Put** This icon toggles the Object Editor's automatic object get/put feature on and off. If it is turned on and you double-click on a frame that contains an image, it will be instantly transferred to the editing screen. If you double-click on the next empty frame (marked with an 'E') after creating a new image, it will be automatically placed into the sprite bank at that position.

## The Grabber Menu

The Grabber Menu is used exclusively to allow you to grab rectangular sections for use as object images from IFF standard picture files.



**Grab Object** If you haven't already loaded (and either packed or kept) an IFF picture file, the 'Grab Object' option will bring up a file requester asking you to select an IFF file to load. Once the image has been loaded, it will be displayed along with a set of cross-hairs. Images are grabbed as rectangular blocks by clicking the mouse pointer at the top left hand corner of the screen and then dragging the mouse pointer away to increase the size of the selected area. Once you're happy that you've selected the object you're interested in, let go of the left mouse button and it will be pulled straight into the Object Editor.



**Put Object** This option works in exactly the same way as the 'Put Object To' icon in the 'bank' menu. It basically allows you to put the image you have grabbed into the sprite bank.



**Load Picture** This option allows you to select the IFF file that you wish the grabber to load. If you turn on either the 'Keep' or 'Pack' options, this will only have to be performed once.



**Grab Palette** By default, this option is turned on. What it does is to instruct the Object Editor to pull in not only the selected object, but also the palette of the picture the selected object was grabbed from.



**Re-load Picture** The ‘Re-load Picture’ toggle tells AMOS that you want to select a new IFF picture every time you click on the ‘Grab Object’ option.



**Pack Picture** If you want to grab several objects from the same picture but your Amiga is a little short on memory, this option will automatically pack the picture that you want to grab from.



**Keep Screen** This instructs the Object Editor to hang onto the picture that you have selected, therefore allowing you to grab several objects from the same picture without having to reload it each time.

## The Hot Spot Menu

All Sprites and Bobs have what is known as a ‘hot spot’ which effectively acts as the object’s ‘handle’ – that is, the point at which AMOS treats as its origin. The Hot Spot menu allows you to change the hot spot setting for any object to any one of nine presets.

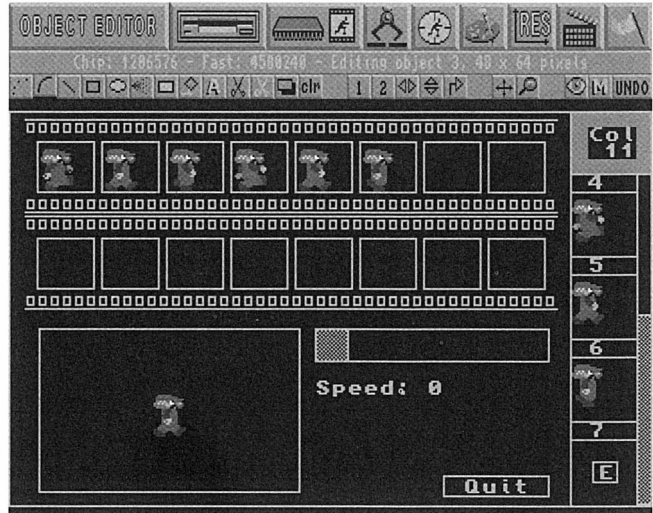


**Auto On/Off** This option allows you to define a default hot spot setting that will be assigned to all objects that you edit from then on. Simply select the hot spot setting that you want and then turn this option on.



**Presets** If the Auto On/Off option is turned off, clicking on one of these nine presets will change the hot spot setting of the object that you are editing to the selected setting.

*The Object Editor's animation editor provides a quick and easy method of testing your sprite and bob animations.*



## The Animation Editor

The Object Editor animation tool allows you to test a sprite or bob's animation by placing the sequence of frames that make up the animation into the Animation Editor. It's very easy to use – all you have to do is to click on the frames in the order that they are to be displayed and AMOS will do the rest. It's worth noting, however, that the animation editor is purely a tool for testing animations and therefore it's up to you to handle the animation manually within your program.





# Object control

- Keeping track of objects with 'data structures'
- Controlling on-screen objects using a joystick or keyboard
- Setting boundaries for object movement
- Creating a 'bouncing' movement
- More advanced object movement patterns

**K**eeping track of two or three objects on the screen is very easy indeed, but if you're writing a particularly frantic arcade game that uses many different objects, chances are that you'll soon find the task of keeping track of them all rather code-intensive. Take a shoot 'em up like the classic 'Asteroids', for example. Because the aliens do not fly in fixed attack waves (which we'll be covering later), it's not possible to simply position each attacker in relation to others. Of course you could simply assign a unique set of variables to each bob. Unfortunately, this approach can complicate matters considerably, since each bob would have to be processed separately.



#### Data structure

When this problem does arise, the simplest method is to take advantage of an age-old games programming concept called the 'Data structure'. As any C or Pascal programmer will tell you, a data structure is essentially a type of array that allows you to group several different variables together under a single heading. AMOS doesn't support real data structures like the C programming language (via the 'struct' command), so we've had to make do with AMOS's still more than capable arrays.

AMOS allows us to create arrays consisting of many different 'dimensions'. A single dimensional array ('Dim Array(Number of Bobs)'), for example) can only hold a single item of information per bob. To be perfectly honest, this isn't really a lot of use as a data structure. Thankfully, you can extend the number of data items that could be held on a single bob simply by adding an extra dimension to the array. For example, 'Dim Array(Number of Bobs, Number of Items)', would give each bob as many number of data items as you wish. Adding an extra dimension to an array effectively tells AMOS that for every element defined by the value of 'Number of Bobs', you want to attach 'n' ('n' is the value defined by 'Number of Items') number of individual items of data.

If, for example, you had 20 bobs, each of which needed 4 items of data attached to it, you want define a data structure to handle all that data using something like this:

```
Dim MyBobs (20,4)
```



Each Bob could be given its own data structure that holds a number of important facts about it – its current X and Y position, Its type (what type of space ship is it?), its bob image number, its direction of movement and perhaps even its status, for example. Using data structures also adds an extra benefit – a simple ‘FOR...NEXT’ loop can be used to control the movements of all Bobs within your game. Although the same code would be used to process the movements of all your bobs, the loop still treats each bob independently of the others. Here’s a bit of skeleton code that demonstrates what I mean:

```
NUMBOBS = 20
Dim MYBOBS(NUMBOBS,4)

For A = 0 to NUMBOBS-1
  Print MYBOBS(A,0) : Rem *** Data item 1
  Print MYBOBS(A,1) : Rem *** Data item 2
  Print MYBOBS(A,2) : Rem *** Data item 3
  Print MYBOBS(A,3) : Rem *** Data item 4
Next A
```



As you can see, the program starts by creating an array that handles 20 bobs, each of which has 4 data items associated with it. These 4 items of data are then extracted by using a FOR...NEXT loop to count from 0 to 19 (don’t forget that the first element in an array is ‘0’ and not ‘1’). The variable ‘A’ is used to define which bob is to be processed during a single loop and the data for that bob is extracted by reading off elements (A,0), (A,1), (A,2) and (A,3).

Most professional games programmers use data structures to keep track of all moving or animated objects within a game. Once you start coding complex arcade games that employ tens of sprites, rather than just the four that our game uses, you’ll soon come to realise that data structures are the only way to control the movement of sprites without your code grinding to a halt. From now on, think data structures!

Before we move onto the next subject in this chapter, why not have a play with this demonstration listing which shows the data structure at its best...

```
Rem *** Handling Multiple Bobs Demonstration
Rem *** Filename - DataStructure.AMOS
```



```
Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0
```

```
Bob Update Off
Double Buffer
Autoback 0
```

```
Load "AMOSBOOK:ABKFiles/Ball.ABK"
Get Sprite Palette
```

```
SPEED=10 : NUMBALLS=16
```

```
Rem *** Initialise 'Balls' data structure
Rem *** BALL(n,0) = Current ball X Position
Rem *** BALL(n,1) = Current ball Y Position
Rem *** BALL(n,2) = Width of bounce
Rem *** BALL(n,3) = Height of bounce
```

```
Dim BALL(NUMBALLS,4)
```

```
For C=0 To NUMBALLS-1
  BALL(C,0)=0
  BALL(C,1)=8+Rnd(200)
  BALL(C,2)=Rnd(SPEED)+1
  BALL(C,3)=Rnd(SPEED)+1
  Set Bob C,1,,
Next C
```

```
Do
  Bob Clear
  For C=0 To NUMBALLS-1
    Rem *** Update position of ball
    BALL(C,3)=BALL(C,3)+1
    BALL(C,0)=BALL(C,0)+BALL(C,2)
```

```
BALL(C,1)=BALL(C,1)+BALL(C,3)

Rem *** Has ball hit ground?
If BALL(C,1)>200
    BALL(C,1)=200
    BALL(C,3)=-BALL(C,3)
End If

Rem *** Has ball reached right hand side
Rem *** of screen?
If BALL(C,0)>300
    BALL(C,2)=-Rnd(SPEED)+1
    BALL(C,3)=Rnd(SPEED)+1
End If

Rem *** Has ball reached left hand side
Rem *** of screen?
If BALL(C,0)<0
    BALL(C,2)=Rnd(SPEED)+1
    BALL(C,3)=Rnd(SPEED)+1
End If

Bob C+1,BALL(C,0),BALL(C,1),1
Next C

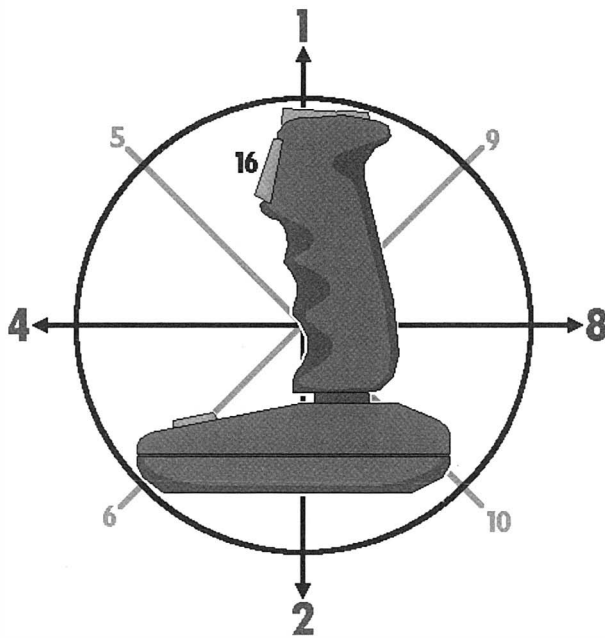
Bob Draw
Screen Swap 0
Wait Vbl
Loop
```

---

## Interactive object control

Although many of the demonstration listings that we've looked at in the last few chapters have made use of the joystick, now is as good a time as any to take a look at how you can use the joystick and even the keyboard to control objects within your own games. Although the joystick is still generally the favoured method of controlling games, you may want to consider offering keyboard control for the benefit of 'purist' game

All digital joysticks can produce only five basic signals – up, down, left, right and fire. AMOS assigns values to these signals and other directions can be detected simply by combining two of these values together.



players that still yearn for the old 8-bit days (many still believe that the keyboard is the only way to control certain types of game!). Better still, using a combination of joystick and keyboard control can add an extra dimension of control to most games – a shoot ‘em up, for example, could be primarily controlled by the joystick but the keyboard could also be used to allow the player to quickly and easily select different weapons.



#### Joystick inputs

Let’s start by taking a look at joystick control. The Amiga offers two control ports, one of which is usually used by the mouse controller. You can, however, read joystick inputs for two-player games from both ports by reading the values returned by the ‘Joy(1)’ and ‘Joy(2)’ functions. Confusingly, ‘Joy(1)’ reads its input from control port 2 (the joystick port) and ‘Joy(2)’ reads its input from control port 1 (the mouse port).

The values returned by these functions are in the form of what is known as a ‘bit pattern’. That is, each direction of movement is assigned its own bit in an 8-bit value and so several joystick movements (up, left and fire, for example), can be read simultaneously.

Don't let the thought of bit patterns worry you, however – because AMOS automatically converts the bit pattern into a decimal number, you can easily translate the values returned into a useable format.

Translating the input from a joystick is very simple indeed, because all the values returned are really nothing more than different combinations of a set of five basic values. If you think about it, a basic digital joystick (all current Amiga joysticks are digital) can only do five things – it can be pushed up, down, left, right or the fire button can be pressed.

All other joystick inputs are nothing more than combinations of these five basic operations – up, left and fire, for example. These five operations have the following values assigned to them.

<b>Joystick up</b>	<b>Joy(1) = 1</b>
<b>Joystick down</b>	<b>Joy(1) = 2</b>
<b>Joystick left</b>	<b>Joy(1) = 4</b>
<b>Joystick right</b>	<b>Joy(1) = 8</b>
<b>Joystick fire</b>	<b>Joy(1) = 16</b>

So, in order to check whether the joystick was being pushed up, all you would do is to compare the value returned by the 'Joy(1)' function to '1'. For more complex joystick inputs, AMOS simply combines the values above to produce other unique values. For example, if the joystick were being pushed up and left with the fire button, the 'Joy(1)' function would return a value of '21' ( $1 + 4 + 16 = 21$ ).

Obviously your code would start to get rather complex if you checked for every possible joystick combination, so the easiest way of keeping track of them all is to make use of the 'AND' logic operation. Here's a very simple demonstration listing:

```
Rem *** Basic Joystick input demonstration
Rem *** Filename - Joystick.AMOS
```

```
Curs Off
```

```
Locate 0,10 : Centre "Move the joystick!"
```





```
Do
  If Joy(1) and 1
    Locate 0,1 : Print "The joystick was pushed up"
  Else
    Locate 0,1 : Print Space$(40)
  End If

  If Joy(1) and 2
    Locate 0,2 : Print "The joystick was pushed down"
  Else
    Locate 0,2 : Print Space$(40)
  End If

  If Joy(1) and 4
    Locate 0,3 : Print "The joystick was pushed left"
  Else
    Locate 0,3 : Print Space$(40)
  End If

  If Joy(1) and 8
    Locate 0,4 : Print "The joystick was pushed right"
  Else
    Locate 0,4 : Print Space$(40)
  End If

  If Joy(1) and 16
    Locate 0,5 : Print "The joystick fire button was pressed!"
  Else
    Locate 0,5 : Print Space$(40)
  End If
Loop
```

If all this messing around with logic operations seems a bit long-winded, then you may want to take advantage of another set of joystick reading functions built into AMOS.

```

RESULT = Jright(PORTNUM)
RESULT = Jleft(PORTNUM)
RESULT = Jup(PORTNUM)
RESULT = Jdown(PORTNUM)
RESULT = Fire(PORTNUM)

```



**PORTNUM** Just like the 'Joy()' function, all five functions can read from either control port by passing either a value of '1' or a '2'. A value of '1' denotes the joystick port (port 2) and a value of '2' denotes the mouse port (port 1).

**RESULT** All five functions return one of two possible results. If a value of '1' is returned, then the joystick is being pushed in the direction of the test. If a value of '0' is returned, however, then the joystick is not being pushed in that direction.

So how do you tie the movement of an object in the joystick? Well, it's very simple indeed. All you have to do is to create two variables that contain the 'X' and 'Y' co-ordinates of the object that you'd like to control with the joystick. Then simply check the status of the joystick and add or subtract a fixed amount from the two co-ordinates. The amount you add or subtract will define the speed of an object's movement, so the larger the amount, the faster the object will move. Here's a quick demonstration:

```

Rem *** Basic Joystick control of an object
Rem *** Filename - JoyMove.AMOS

```



```

Screen Open 0,320,256,4,Lowres
Flash Off : Curs Off : Cls 0

```

```

Load "AMOSBOOK:AbkFiles/SpaceInvader.ABK"
Get Sprite Palette

```

```

Paper 0 : Pen 1
Locate 0,10 : Centre "Move me with the joystick!"

```

```
SPRX=160
SPRY=128
FRAME=1

Do
  If Jup(1)=-1
    SPRY=SPRY-2
  End If

  If Jdown(1)=-1
    SPRY=SPRY+2
  End If

  If Jleft(1)=-1
    SPRX=SPRX-2
  End If

  If Jright(1)=-1
    SPRX=SPRX+2
  End If

  If Fire(1)=-1
    Boom
    FRAME=2
  Else
    FRAME=1
  End If

  Bob 1,SPRX,SPRY,FRAME
  Wait Vbl
Loop
```

## Keyboard control

Although the joystick is undoubtedly the most popular method of controlling the player's sprite within a game, some games still lend themselves particularly well to keyboard control. Many game players find keyboard control favourable because it offers far greater precision than the rather haphazard results returned by your average joystick. Platforms games are a good example – positioning the player's sprite in exactly the position necessary to jump between platforms is considerably easier when the game is played with the keyboard. Even games that don't obviously need keyboard control can benefit too – a shoot 'em up, for example, could use certain keys on the Amiga keyboard to give the player the ability to change weapons, drop a 'smart' bomb or even activate a cheat mode.



**Extra control**

The simplest form of keyboard input offered by AMOS is the 'Input' command, but this has the distinct disadvantage of stopping program executing until the user presses the 'RETURN' key – hardly an ideal solution in a fast-paced arcade game! The answer lies in what the techies call 'Scan codes' – these are special codes generated by the Amiga's keyboard hardware that can easily be read 'on the fly' without halting program execution. Each and every key on the Amiga's keyboard has its own unique scan code value and – unlike 'ascii' codes, they never change, regardless of the type of keymap setting the user has assigned to their keyboard.



**Scan codes**

In order to read the scancode of a particular key, you need to start by calling the 'MYKEY\$=Inkey\$' function, which checks for a key depression and stores the key character ('A', 'a', 'B', 'd' etc) into a string variable (in this case, the string variable is called 'MYKEY\$').

The result returned by this function is of no interest to us, but it's vitally important that this function is called in order for the scan code for that key to be read. Once the 'Inkey\$' function has been called, the scan code for the key that the user pressed is held internally and you can then read it into a variable using the following command:

---

**SCANCODEVALUE = ScanCode**



This command would put the scan code value of the key that the user pressed into an integer variable called 'SCANCODEVALUE'. The values returned by the 'Scan Code' function do not cover the extra 'control keys' offered by the Amiga (Left and right shift and the two Amiga keys, for example), because they do not have scan codes associated with them. In order to detect them, you need to check the status of them all using the following command:

---

**CODE = Key Shift**



The value returned by the 'Key Shift' function is in the form of a 'bit pattern' (remember those from our coverage of the 'Joy()' function!) and so several control keys can be detected simultaneously by combining these eight basic values.

<b>Left Shift</b>	<b>Key Shift = 1</b>
<b>Right Shift</b>	<b>Key Shift = 2</b>
<b>Caps Lock</b>	<b>Key Shift = 4</b>
<b>Ctrl</b>	<b>Key Shift = 8</b>
<b>Left Alt</b>	<b>Key Shift = 16</b>
<b>Right Alt</b>	<b>Key Shift = 32</b>
<b>Left Amiga</b>	<b>Key Shift = 64</b>
<b>Right Amiga</b>	<b>Key Shift = 128</b>

If, for example, you wanted to check to see whether the 'Right Shift' and 'Right Amiga' keys were being pressed simultaneously, all you'd have to do is to compare the value returned by the 'Key Shift' function with '130' ( $2 + 128 = 130$ ).

Here's a short listing that displays both the scan codes and 'key shift' status of control keys. You'll find this program useful for getting the scan code values for particular keys:

```
Rem *** Keyboard Scan Code Reader
Rem *** Filename - ScanCodeReader.AMOS
```



```
Curs Off
```

```
Rem *** Clear Keyboard buffer
Clear Key
```

```
Do
```

```
  A$=Inkey$
  KEY=Scancode
  CTRLKEY=Key Shift
```

```
  If KEY<>0
    Locate 1,1 : Print "Key Scancode = ";KEY;" "
  End If
  If CTRLKEY<>0
    Locate 1,2 : Print "Control Key = ";CTRLKEY;" "
  End If
```

```
Loop
```



You could quite easily use the ‘Scancode’ function to detect certain keys, but it’s not that fast. The ‘Key State()’ function is much faster. It simply monitors the status of a single key and you don’t even have to call the ‘Inkey\$’ function first to make it work! It has the following format:

```
RESULT = Key State(SCANCODE)
```



**SCANCODE** The scan code for the key that you want the function to monitor. For example, feeding it a value of ‘49’ will instruct it to monitor the ‘Z’ key. A full list of scan codes is given overleaf, but the ‘Scan Code Reader’ program above will provide you with the scan codes you require – just run the program and press the key that you are interested in.

**RESULT** This function returns one of two possible values. If the key was being pressed when the test was made, a value of ‘-1’ will be returned. If the key wasn’t being pressed, however, a value of ‘0’ will be returned.



Amiga keyboard scan codes (above).

Applying all this theory to control the movements of an on-screen object is very simple indeed. Below is a short demonstration listing that uses both functions to move a Bob around the screen:

```
Rem *** Basic Keyboard control of an object
Rem *** By Jason Holborn
```



```
Screen Open 0,320,256,4,Lowres
Flash Off : Curs Off : Cls 0
```

```
Load "AMOSBOOK:AbkFiles/SpaceInvader.ABK"
Get Sprite Palette
```

```
Paper 0 : Pen 1
Locate 0,10 : Centre "Use 'Z','X' and '#','/' keys to move me"
Locate 0,11 : Centre "Press 'Right Shift' to fire!"
```

```
SPRX=160
SPRY=128
FRAME=1
```

```
Do
  Rem *** Check for '#' key...
  If Key State(42)=-1
    SPRY=SPRY-2
  End If

  Rem *** Check for '/' key...
```

```
If Key State(58)=-1
    SPRY=SPRY+2
End If

Rem *** Check for 'Z' key...
If Key State(49)=-1
    SPRX=SPRX-2
End If

Rem *** Check for 'X' key...
If Key State(50)=-1
    SPRX=SPRX+2
End If

If Key Shift=2
    BocoM
    FRAME=2
Else
    FRAME=1
End If

Bob 1,SPRX,SPRY,FRAME
Wait Vbl

Loop
```

---

## Restricting object movement

Simply increasing or decreasing the values of an object's 'X' and 'Y' co-ordinates every vertical blank will cause an object to move, but it's also important to keep track of the object's movement so that it doesn't disappear off of the screen completely, never to be seen again. Although you may want this to happen for certain objects (an attack wave in a shoot 'em up, for example), the player's sprite must always be kept within the boundaries of the current screen.

Thankfully, this is very easy to do. All you need to do is to keep track of an object's 'X' and 'Y' co-ordinates and when they drop below or rise above a set of minimum and maximum values (0 and 320 for horizontal



movement on a low resolution screen, for example), you simply reset them to the appropriate value. If, for example, the player moved the sprite to the far left of the screen and the 'X' co-ordinate for that object dropped below 0 (say, for example, it dropped to '-4'), all you do is reset it to '0'. So, no matter how hard the player tried to move the object off of the screen, the 'X' co-ordinate of that object would never be allowed to drop below '0' once the object was drawn onto the screen. Here's a short demonstration that puts that theory into practice:

```

Rem *** 'Boundary Restricted' object movement
Rem *** Filename - BoundaryRestricted.AMOS

Screen Open 0,320,256,4,Lowres
Flash Off : Curs Off : Cls 0

Load "AMOSBOOK:AbkFiles/Ball.ABK"
Get Sprite Palette

Bob Update Off
Double Buffer
Autoback 0

XSIZE=16 : YSIZE=16 : Rem *** X and Y size of Bob
SPRX=160 : SPRY=128

Rem *** Define boundary
XMAX=320-XSIZE : XMIN=0
YMAX=256-YSIZE : YMIN=0

Do
  Bob Clear

  If Joy(1) and 1
    SPRY=SPRY-2
    If SPRY<YMIN
      SPRY=YMIN
    End If

```



```
End If
If Joy(1) and 2
    SPRY=SPRY+2
    If SPRY>YMAX
        SPRY=YMAX
    End If
End If
If Joy(1) and 4
    SPRX=SPRX-2
    If SPRX<XMIN
        SPRX=XMIN
    End If
End If
If Joy(1) and 8
    SPRX=SPRX+2
    If SPRX>XMAX
        SPRX=XMAX
    End If
End If

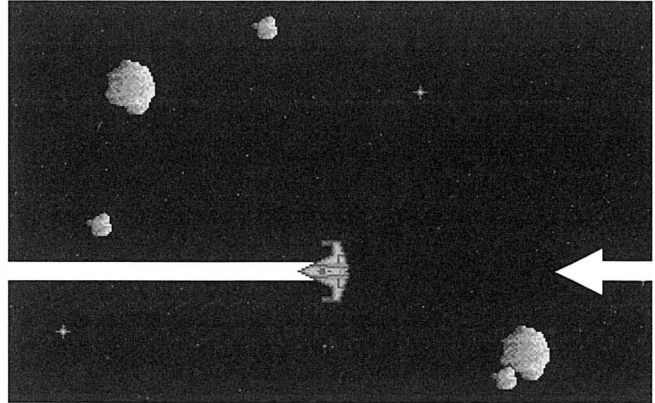
Bob 1, SPRX, SPRY, 1

Bob Draw
Screen Swap 0
Wait Vbl
Loop
```

If you're writing a game like the classic 'Asteroids', it's often useful to be able to 'wrap' a bob around so that when it disappears off of any one of the four edges of the screen, it appears on the opposite side. Other games that have used this technique include 'Bubble Bobble' (if Bub or Bob – or indeed the ghosts that chase them – dropped off the bottom of the screen, they would instantly reappear at the top) and even – to a lesser extent – the infamous classic PacMan.

Achieving this effect in AMOS is very simple indeed. All you have to do is to continuously check the 'X' and 'Y' co-ordinates of your objects and when they drop below or rise above a set of minimum and maximum

*For games like the classic 'Asteroids', the objects 'wrap around' the screen so that when they disappear off of one side, they reappear on the other side.*



values (0 and 320 for horizontal movement on a low resolution screen, for example), simply subtract the appropriate co-ordinate from the width or the height of the screen. If, for example, the 'X' co-ordinate for the object dropped below 0 (let's say it was set to '-4'), subtracting it from 320 would give you a new 'X' co-ordinate of 316, therefore effectively causing it to re-appear on the opposite side of the screen. If, on the other hand, it rose above 320 (say '324'), subtracting it from 320 would give you a new 'X' co-ordinate of '-4'. However, in order to stop it from wrapping back around again, it should – in theory at least – rise above the minimum value by the time it is checked again because it would be updated again by the joystick movement routine. Try this demonstration:

```
Rem *** 'Wrap around' object movement
Rem *** Filename - WrapAround.AMOS
```

```
Screen Open 0,320,256,4,Lowres
Flash Off : Curs Off : Cls 0
```

```
Load "AMOSBOOK:AbkFiles/Ball.ABK"
Get Sprite Palette
```

```
Bob Update Off
Double Buffer
Autoback 0
```



```
SPRX=160 : SPRY=128

Do
  Bob Clear

  If Joy(1) and 1
    SPRY=SPRY-2
  End If
  If Joy(1) and 2
    SPRY=SPRY+2
  End If
  If Joy(1) and 4
    SPRX=SPRX-2
  End If
  If Joy(1) and 8
    SPRX=SPRX+2
  End If

  Rem *** Check object X and Y co-ordinates
  If SPRX>320 or SPRX<0
    Rem *** Wrap around X axis
    SPRX=320-SPRX
  End If
  If SPRY>256 or SPRY<0
    Rem *** Wrap around Y axis
    SPRY=256-SPRY
  End If

  Bob 1, SPRX, SPRY, 1

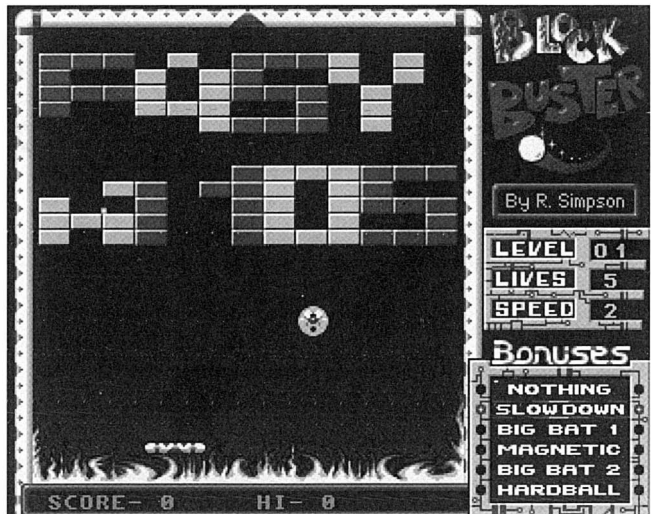
  Bob Draw
  Screen Swap 0
  Wait Vbl
Loop
```

## 'Bouncing' a bob

So far we've looked at how to restrict the movement of a bob within a predefined boundary and how to make a bob 'wrap around' to the opposite side of the screen when it goes beyond those boundaries. The only type of bob movement control routine left that I can think of is the classic 'Break Out'-style bouncing movement, where an object bounces off the edges of a screen. Obviously this only really works if the object is constantly moving (an object can't bounce if it stops whenever it hits a boundary!), so it's only really of use when the object is either computer-controlled or is never allowed to stop moving.

The key to this little routine is the way both the object's 'X' and 'Y' coordinates and direction of movement are stored together in a data structure. With each loop of the program, the direction of movement along both the 'X' and 'Y' axis are added to the object's co-ordinates. If the object strikes the left or right hand side of the screen, the object's direction of movement along the 'X' axis is reversed and if the object strikes the top or bottom edges of the screen, the direction of movement along the 'Y' axis is reversed. Obviously this routine doesn't allow you to change the angle of the object's movement, but you should find it more than suitable for most 'Break Out'-style games.

*Games like 'Block Buster', Ronny Simpson's brilliant 'Break Out' clone restrict the movement of an object by 'bouncing' it off the edges of the boundary that contains it.*





```
Rem *** Bouncing Bob Demonstration
Rem *** Written by Jason Holborn
```

```
Screen Open 0,320,256,4,Lowres
Flash Off : Curs Off : Cls 0
Double Buffer
Autoback 0
Bob Update Off
```

```
Load "AMOSBOOK:AbkFiles/Ball.ABK"
Get Sprite Palette
```

```
Rem *** Initialise Ball data structure
```

```
Dim BALL(4)
```

```
BALL(0)=160 : Rem *** X Position of Ball
BALL(1)=128 : Rem *** Y Position of Ball
BALL(2)=5 : Rem *** X Direction of Ball
BALL(3)=5 : Rem *** Y Direction of Ball
```

```
Hot Spot 1,$11
Set Bob 1,1,,
```

```
Do
```

```
    Bob Clear
```

```
    Rem *** Update Ball position
    BALL(0)=BALL(0)+BALL(2)
    BALL(1)=BALL(1)+BALL(3)
```

```
    Rem *** Has ball hit top or bottom edge of screen?
    If BALL(0)>310 or BALL(0)<20
        BALL(2)=-BALL(2)
    End If
```

```
    Rem *** Has ball hit left or right edge of screen?
```

```
If BALL(1)>246 or BALL(1)<20
    BALL(3)=-BALL(3)
End If
```

```
Bob 1,BALL(0),BALL(1),1
Bob Draw
Screen Swap 0
Wait Vbl
```

```
Loop
```

---

## Advanced object movement

Getting a bob or sprite onto the screen and then moving it is a very simple aspect of AMOS programming, but the real challenge comes when you want to apply what you've learned to 'real' programming projects like games and demos. So far all the examples that we've covered have assumed that all we want to do is to simply move objects in the four basic directions – up, down, left and right. Whilst this may be fine for basic shoot 'em ups, very few games or demos employ such simplistic movement of objects.

Writing the sort of object movement routines that you find in arcade games is not quite as simple as it may first appear. For example, have you ever sat down and considered how much work is involved in making an object like the infamous Nintendo plumber 'Super Mario' jump from one level to another? Believe it or not, calculating the jump involves a lot of complex mathematics using trigonometric functions such as Cosine and Sine! And how do you make an object move in any of the 360 degrees of movement? Once again, some complex mathematics are involved. And you thought games programmers had it easy!

Thankfully, you don't have to worry about coding such complex routines yourself because I've written them all for you! Over the next few pages you'll find a whole range of game and demo-related routines that cover just about every conceivable way of moving a sprite or bob. If you can think of a particularly obscure way of moving a bob, then chances are that there's a routine somewhere within this chapter that will do the job! I've tried to cover as many different types of object movement that I

could possibly think of. Attack waves, 360 degree movement, making an object jump – they're all here plus a few more besides.

## 360 degree movement

Simply adding or subtracting a fixed value depending upon the direction of a joystick is fine for basic object control, but it does have the disadvantage of producing rather rigid-looking movements – up, down, up and left, right etc. Unless you're lucky enough to have an analogue joystick (and you know how to read the values that it returns!), eight basic directions are all that standard Amiga joysticks allow. Whilst this is fine for your average shoot 'em up and platform game romp, real objects just don't behave that way. In the real world, any object is capable of turning and moving in a full 360 degrees of movement.

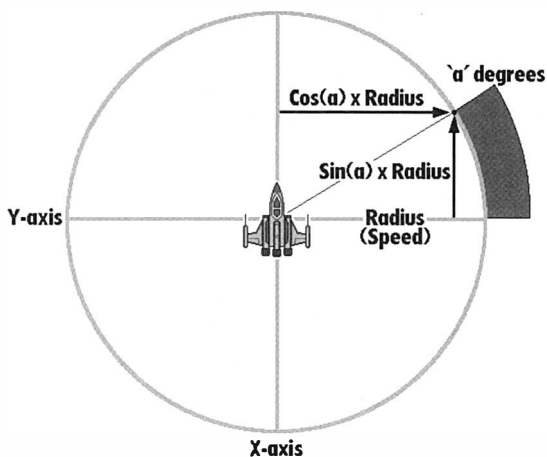
You can emulate this through AMOS by taking advantage of AMOS's powerful mathematical functions,  $\text{Cos}()$  and  $\text{Sin}()$ , to calculate the direction of movement of an object in any direction.



### AMOS's maths functions

It's worth noting however, that in order to use these functions, you must have the file 'mathtrans.library' in the 'LIBS' directory of your boot disk. Bear this in mind if you intend to produce a game that runs from its own disk – even once compiled, AMOS programs still expect to find this file! The routine works by increasing or decreasing a variable that contains

*Objects can easily be moved in 360 degrees of movement using AMOS' powerful trigonometric functions 'Cos()' and 'Sin()'.*





the 'angle' of the object whenever the joystick is pushed left or right – '0' degrees denotes a movement to the right and the angle variable would increase to 360 in an anti-clockwise direction. Whenever the object is moved, the direction along the 'X' axis is calculated using the formula 'Cos(angle) \* speed' and the direction along the 'Y' axis is calculated using the formula 'Sin(angle) \* speed'. Note how the 'Degree' command tells AMOS to perform its calculations using degrees rather than radians.

```
Rem *** 360 degree object movement
Rem *** Filename - 360DegreeMovement.AMOS
```



```
Screen Open 0,320,256,4,Lowres
Flash Off : Cls 0 : Curs Off
```

```
Load "AMOSBOOK:AbkFiles/Ball.ABK"
Get Sprite Palette
```

```
Bob Update Off
Double Buffer
Autoback 0
```

```
Rem *** Define Ball's data structure
Dim OBJ#(4)
```

```
OBJ#(0)=160 : Rem *** Ship X Position
OBJ#(1)=128 : Rem *** Ship Y Position
OBJ#(2)=0 : Rem *** Speed
OBJ#(3)=0 : Rem *** Angle (0 = North)
```

```
Degree
```

```
Do
```

```
    Bob Clear
```

```
    Rem *** Update speed of object
    If Joy(1) and 2
        OBJ#(2)=OBJ#(2)-0.1
```

```
        If OBJ#(2)<0
            OBJ#(2)=0
        End If
    End If
    If Joy(1) and 1
        OBJ#(2)=OBJ#(2)+0.1
        If OBJ#(2)>10
            OBJ#(2)=10
        End If
    End If

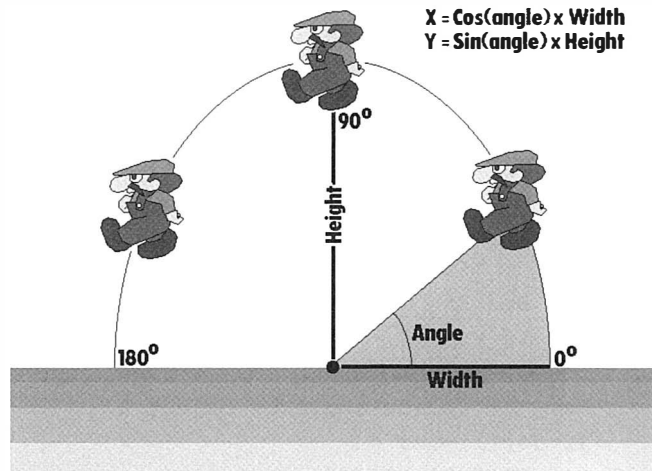
    Rem *** Update angle of object
    If Joy(1) and 4
        OBJ#(3)=OBJ#(3)-OBJ#(2)/2
    End If
    If Joy(1) and 8
        OBJ#(3)=OBJ#(3)+OBJ#(2)/2
    End If

    Rem *** Calculate new bearing
    OBJ#(0)=OBJ#(0)+Sin(OBJ#(3))*OBJ#(2)/2
    If OBJ#(0)>320 or OBJ#(0)<0
        OBJ#(0)=320-OBJ#(0)
    End If
    OBJ#(1)=OBJ#(1)+Cos(OBJ#(3))*OBJ#(2)/2
    If OBJ#(1)>256 or OBJ#(1)<0
        OBJ#(1)=256-OBJ#(1)
    End If

    Bob 1,OBJ#(0),OBJ#(1),1
    Plot OBJ#(0),OBJ#(1)

    Bob Draw
    Screen Swap 0
    Wait Vbl
Loop
```

The same 'Cos()' and 'Sin()' functions that we used to move an object in 360 degrees of movement are used to make an object jump.



## Making a bob 'jump'

If you're writing a game where you need to make an object appear to jump from one place to another, then this routine is for you. What it does is to calculate the points required to smoothly move an object through a semi-circle using the same 'Sin()' and 'Cos()' functions that we used to produce the 360 degree movement routine above.

In many respects, this routine is actually considerably simpler, because all it does is to calculate a invisible semi-circle which is split into a series of points which the object is moved through. I've written it in such a way that the procedure '\_JUMP' can easily be pulled out and imported into your own games software with minimal changes needing to be made.

Once again, a data structure is created for the object that contains two very important elements – the object's status (is he walking or jumping?) and the current angle of the jump. If the object isn't moving, then the angle is ignored and the status is given a value of '1'. If the player then presses the fire button, the status is changed to '0' so that all joystick input is ignored until the jump is finished (once in the air, the object should not be allowed to change direction!). The angle of the jump is then used to calculate the current position of the object using the 'Sin()' and 'Cos()' functions. A 'speed' variable is also used to calculate the speed of the jump. The value held in this variable is simply added to the

angle each time a new position for the object is calculated and when the angle increases above 180, the jump is finished and the object's status is reset to '1' allowing it to move under joystick control again.

Below is a listing that demonstrates this movement technique in action. Note how the height, width and the speed of the jump can easily be adjusted from within the '\_JUMP' procedure! Believe it or not, you now have one of the most complex routines needed for writing platform games!

```

Rem *** Bob 'Jump' Demonstration
Rem *** Filename - BobJump.AMOS

Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0

Bar 0,200 To 320,256

Bob Update Off
Double Buffer : Autoback 0

Load "AMOSBOOK:AbkFiles/Mario.abk"
Get Sprite Palette

Rem *** Define Mario Bob data structure

Dim MARIO(4)
MARIO(0)=160 : Rem *** Bob X Position
MARIO(1)=200 : Rem *** Bob Y Position
MARIO(2)=0 : Rem *** Status 0=Walk 1=Jump
MARIO(3)=0 : Rem *** Jump angle

For C=1 To 6
    Hot Spot C,$12
Next C

FRAME=1 : FRAMEDELAY=0 : SPEED=4

```



```
Global DIRECTION, XOFFSET, YOFFSET, MARIO(), SPEED
```

```
Do
```

```
  Bob Clear
```

```
  If Joy(1) and 4
```

```
    If MARIO(2)=0
```

```
      MARIO(0)=MARIO(0)-SPEED
```

```
      DIRECTION=-1
```

```
      FRAME=FRAME or $8000
```

```
    End If
```

```
  End If
```

```
  If Joy(1) and 8
```

```
    If MARIO(2)=0
```

```
      MARIO(0)=MARIO(0)+SPEED
```

```
      DIRECTION=1
```

```
      FRAME=FRAME and %111
```

```
    End If
```

```
  End If
```

```
  If Joy(1)=0
```

```
    If MARIO(2)=0
```

```
      DIRECTION=0
```

```
      FRAME=FRAME and $8000
```

```
      FRAME=FRAME+3
```

```
    End If
```

```
  End If
```

```
  If Joy(1) and 16
```

```
    If MARIO(2)=0
```

```
      MARIO(2)=1
```

```
      XOFFSET=MARIO(0)
```

```
      YOFFSET=MARIO(1)
```

```
    End If
```

```
  End If
```

```
  If MARIO(2)=1
```

```
    _JUMP
```

```
    FRAME=FRAME and $8000
```

```
    FRAME=FRAME+3
```

```
  End If
```

```
If MARIO(0)>320
    MARIO(0)=320
End If
If MARIO(0)<0
    MARIO(0)=0
End If

Bob 1,MARIO(0),MARIO(1),FRAME

Bob Draw
Screen Swap 0
Wait Vbl

If FRAMEDELAY>6-SPEED
    FRAMEDELAY=0
    FRAME=FRAME+1
    If FRAME=7 or FRAME=$8007
        FRAME=FRAME and $8000
        FRAME=FRAME+1
    End If
End If
FRAMEDELAY=FRAMEDELAY+1
Loop

Procedure _JUMP
    HEIGHT=160 : Rem *** Maximum height of Jump
    WIDTH=80 : Rem *** Width of Jump
    JUMPSPEED=SPEED*2 : Rem *** Speed of Jump

Degree

If MARIO(3)<181

    Rem *** Jump left
    If DIRECTION=-1
        X=Cos(MARIO(3))*WIDTH/2
        MARIO(0)=XOFFSET+X-WIDTH/2
```

```

End If

Rem *** Jump Right
If DIRECTION=1
    X=-Cos(MARIO(3))*WIDTH/2
    MARIO(0)=XOFFSET+X+WIDTH/2
End If

Y=-Sin(MARIO(3))*HEIGHT/2

MARIO(1)=YOFFSET+Y
MARIO(3)=MARIO(3)+JUMPSPEED
Else
    Rem *** Reset angle and status
    MARIO(1)=YOFFSET
    MARIO(2)=0
    MARIO(3)=0
End If
End Proc

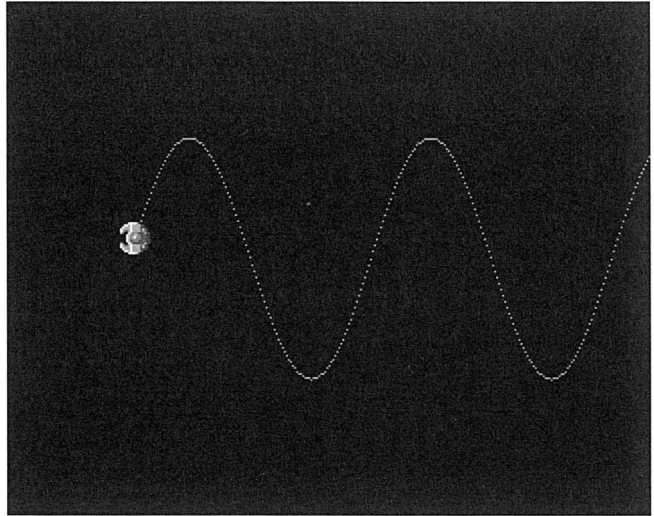
```

## Attack waves

If you're writing arcade games of the shoot 'em up variety then one of the most important aspects of the game's design is to create attack waves for all the marauding aliens that will challenge the player's gaming abilities. Most would-be games programmers tend to stick with the rather boring 'straight line' approach that simply moves your aliens across the screen without changing their direction. Although this can be effective at times, just about any games player worth their salt will eventually realise that the game can easily be conquered simply by being in the right place at the right time.

A far more challenging approach is to make use of AMOS's powerful math functions to calculate the route of an alien 'on the fly'. Most commercial shoot 'em ups use the math approach simply because it allows the programmer to produce complex sprite and bob movements with a minimal amount of hassle. The keys to all this digital trickery are the AMOS 'Sin()' and 'Cos()' functions that, as any mathematician will tell you, are used to calculate the sine and cosine values of a given value.

Complex attack waves for shoot 'em ups can easily be created using the 'Sin()' function.



Note that in order to use the values returned by either, you should include the 'Degree' command near the start of your program. This tells AMOS to convert the 'radian' values returned by AMOS's math functions to far more manageable 'degrees'. This listing moves an spaceship along a series of sine waves, drawing the waves to show you the movement path:

```
Rem *** Sine Attack Wave Demonstration
Rem *** Filename - SineAttack.AMOS

SPRX=0 : SPRY=0 : FRAME=8 : FRAMEDELAY=0

Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0

Load "AMOSBOOK:AbkFiles/SpaceShip.abk"
Get Sprite Palette

Bob Update Off
Double Buffer : Autoback 0

HEIGHT=60 : Rem *** Height of wave
```





```
WIDTH=1 : Rem *** Width of wave
POSITION=128 : Rem *** Centre POSITION OF Wave
SPEED=1 : Rem *** Speed of bob movement
```

```
Ink 5
Degree
```

```
Do
```

```
  For SPRX=320 To -40 Step -SPEED
    Bob Clear
```

```
    Rem *** Calculate Bob position
    SPRY=Sin(SPRX*WIDTH)*HEIGHT+POSITION
```

```
    Bob 1,SPRX-8,SPRY-8,FRAME
    Plot SPRX,SPRY
```

```
    Bob Draw
    Screen Swap 0
    Wait Vbl
```

```
    Rem *** Update animation
    If FRAMEDELAY=5
      FRAME=FRAME-1
      If FRAME=0
        FRAME=8
      End If
      FRAMEDELAY=0
```

```
    End If
    FRAMEDELAY=FRAMEDELAY+1
```

```
  Next SPRX
```

```
  WIDTH=WIDTH+1
```

```
  If WIDTH=5
    WIDTH=1
```

```
  End If
```

```
Loop
```

What's more, if you want to generate an attack wave that flies in sequence (just like the Red Arrows), you don't even have to calculate a new position for each bob by calling the 'Sin()' or 'Cos()' functions over and over again. The simplest way is to simply have a single set of co-ordinates (the 'X' and 'Y' positions of the 'central' bob) and then simply calculate the positions of all bobs around this single set of co-ordinates simply by 'offsetting' them. Here's a slightly modified version of the same listing:

```
Rem *** 'V' Shape Sine Attack Wave Demonstration
Rem *** Filename - V-AttackWave.AMOS
```



```
SPRX=0 : SPRY=0 : FRAME=8 : FRAMEDELAY=0
```

```
Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0
```

```
Load "AMOSBOOK:AbkFiles/SpaceShip.abk"
Get Sprite Palette
```

```
Bob Update Off
Double Buffer : Autoback 0
```

```
HEIGHT=60
WIDTH=1
POSITION=128
SPEED=2
```

```
Ink 5
Degree
```

```
Do
  For SPRX=320 To -40 Step -SPEED
    Bob Clear

    Rem *** Calculate Bob position
    SPRY=Sin(SPRX*WIDTH)*HEIGHT+POSITION
```

```

Bob 1, SPRX, SPRY, FRAME
Bob 2, SPRX+20, SPRY-30, FRAME
Bob 3, SPRX+20, SPRY+30, FRAME
Bob 4, SPRX+40, SPRY-60, FRAME
Bob 5, SPRX+40, SPRY+60, FRAME
Bob 6, SPRX+40, SPRY, FRAME

```

```

Bob Draw
Screen Swap 0
Wait Vbl

```

```

Rem *** Update animation
If FRAMEDELAY=5
    FRAME=FRAME-1
    If FRAME=0
        FRAME=8
    End If
    FRAMEDELAY=0
End If
FRAMEDELAY=FRAMEDELAY+1

```

```
Next SPRX
```

```

WIDTH=WIDTH+1
If WIDTH=5
    WIDTH=1
End If

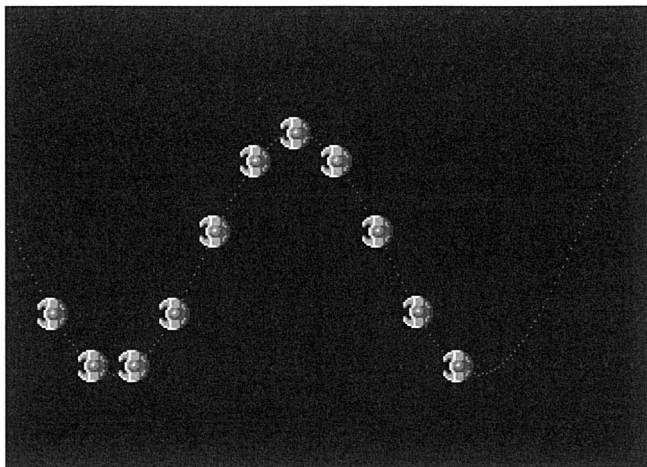
```

```
Loop
```

### **'Snake' attack waves**

The only real problem with this approach is that the attack wave looks rather 'wooden' and therefore it lacks the sort of fluidity that you find in most commercial arcade games. If you want to create an attack wave with each ship following in the other's path (forming a sort of 'snake' formation), then all you have to do is to calculate a new 'Y' co-ordinate for each ship by offsetting the 'X' co-ordinate of the ship by a fixed number of units. The great thing about this routine is that you really only need to keep track of the 'X' co-ordinate of a single ship – because the

More complex attack waves can be generated simply by calculating the position of each alien separately on the sine wave.



positions of all the other ships are simply calculated by offsetting the 'X' co-ordinate of the first ship, there's no need to waste valuable memory. Fancy a demonstration? Well, as if by magic, here's a demonstration listing that I prepared earlier (Blue Peter eat your heart out!).

```
Rem *** Sine 'Snake Wave' Demonstration
Rem *** Filename - SnakeWave.AMOS
```



```
SPRX=0 : SPRY=0 : FRAME=8 : FRAMEDELAY=0
```

```
Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0
```

```
Load "AMOSBOOK:AbkFiles/SpaceShip.abk"
Get Sprite Palette
```

```
Bob Update Off
Double Buffer : Autoback 0
```

```
HEIGHT=60
WIDTH=1
POSITION=128
SPEED=2
```

Degree

Ink 5

Do

```
For SPRX=319 To -200 Step -SPEED
```

```
Bob Clear
```

```
Rem *** Calculate new 'Y' co-ordinate
```

```
Rem *** for all objects
```

```
OBJNUMBER=0
```

```
For C=0 To 200 Step 20
```

```
Y=Sin((SPRX+C)*WIDTH)*HEIGHT+POSITION
```

```
Bob OBJNUMBER,SPRX+C,Y,FRAME
```

```
OBJNUMBER=OBJNUMBER+1
```

```
Next C
```

```
Bob Draw
```

```
Screen Swap 0
```

```
Wait Vbl
```

```
Rem *** Update animation
```

```
If FRAMEDELAY=5
```

```
FRAME=FRAME-1
```

```
If FRAME=0
```

```
FRAME=8
```

```
End If
```

```
FRAMEDELAY=0
```

```
End If
```

```
FRAMEDELAY=FRAMEDELAY+1
```

```
Next SPRX
```

```
WIDTH=WIDTH+1
```

```
If WIDTH=5
```

```
WIDTH=1
```

```
End If
```

Loop

## Circular attack waves

Some very fancy attack waves can be produced by combining both the 'Sin()' and 'Cos()' functions. As you will probably already know, performing a loop that calculates and then plots the Cosine and Sine values between 0 and 360 will produce a perfect circle, and you can use this to produce some very nice looking attack waves. Type in this listing for a vivid example – This code too draws the paths of the objects, so you'll be able to see the effect of each 'Sin()' and 'Cos()' calculation. Who said trigonometry was useless!

```
Rem *** Complex Sine and Cosine Attack Wave Example
Rem *** Filename - CosineSineAttack.AMOS
```



```
FRAME=8 : FRAMEDELAY=0
```

```
Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0 : Hide
```

```
Load "AMOSBOOK:Abkfiles/SpaceShip.abk"
Get Sprite Palette
```

```
Bob Update Off
Double Buffer : Autoback 0
```

```
XPOSITION#=360
YPOSITION=128
SIZE=50
```

```
Degree
Do
```

```
  For C=0 To 360 Step 4
    Bob Clear
```

```
    X1=Cos(C)*SIZE+XPOSITION#
    X2=Cos(C-90)*SIZE+XPOSITION#
    Y1=Sin(C)*SIZE+YPOSITION
    Y2=Sin(C-90)*SIZE+YPOSITION
```

```
Bob 1,X1,Y1,FRAME
Bob 2,X2,Y1,FRAME
Bob 3,X1,Y2,FRAME
Bob 4,X2,Y2,FRAME
```

```
Ink 2 : Plot X1,Y1
Ink 3 : Plot X1,Y2
Ink 5 : Plot X2,Y1
Ink 6 : Plot X2,Y2
```

```
Bob Draw
Screen Swap 0
Wait Vbl
```

```
Rem *** Note how a 'real' number is used to
Rem *** slow down movement! AMOS simply rounds
Rem *** these numbers down for the 'Bob' command
```

```
XPOSITION#=XPOSITION#-0.5
```

```
If XPOSITION#<-20
    XPOSITION#=360
```

```
End If
```

```
FRAMEDELAY=FRAMEDELAY+1
```

```
If FRAMEDELAY=5
    FRAMEDELAY=0
    FRAME=FRAME-1
    If FRAME=0
        FRAME=8
```

```
End If
```

```
End If
```

```
Next C
```

```
Loop
```

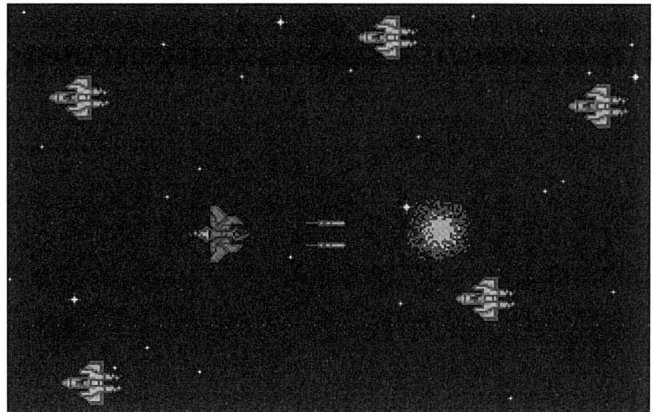
## Firing missiles

What makes shoot ‘em up games so appealing is the chance to vent all that fury and frustration that builds up inside us all during the day on some poor hapless alien that just happens to have wandered in front of your laser cannons. All shoot ‘em ups and quite a few other game genres allow the player’s sprite (and perhaps even the aliens that you’re attacking) to fire objects at other sprites in the hope of bringing about their untimely demise. In a game like ‘Xenon 2’, for example, the ability to fire projectiles at other sprites is taken to extremes with a whole range of alien-splattering hardware available (including the infamous ‘Super Nashwan Power’ weaponry).

If you’re writing a shoot ‘em up, then the routine below will definitely be useful to you. It displays a small spaceship on the screen that can be moved around with the joystick. When you press the firebutton, however, a pair of missiles shoots from the front of the ship and flies across the screen. Give it a try – I’ve also added a sampled sound effect which helps to give the program a little bit of extra atmosphere.

Once again, the missiles are handled internally using a set of four data structures, one for each missile. When the player presses the firebutton, the ‘\_FIRE’ routine checks to see whether a missile is available for firing (all four missiles could still be on the screen!). If there is, the missile is effectively fired by setting the missile’s status flag, ‘MISSILE(n,0)’ – ‘n’

*If you’re writing a shoot ‘em up, then the player’s ship must be able to fire missiles at the attacking aliens.*





is the number of the missile – to 1. The starting position of the missile is then calculated by offsetting it from the position of the spaceship. Any missiles that have been fired are then constantly moved until they leave the screen. Once a missile has left the screen, it's then effectively available for refiring. Clever, eh?

```
Rem *** Firing Missiles Demonstration
Rem *** Filename - FireMissile.AMOS
```



```
Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0 : Hide
Double Buffer : Autoback 0
Bob Update Off : Sprite Update Off
```

```
Load "AMOSBOOK:AbkFiles/CosmoSoundFX.ABK"
Load "AMOSBOOK:AbkFiles/CosmoShips.ABK"
Get Sprite Palette
```

```
Rem *** Initialise ship Bob...
SPRX=40 : SPRY=128
```

```
Rem *** Initialise Missiles Data structure....
Rem *** Missiles(n,0) = Missile status 0=Ready 1=Fired
Rem *** Missiles(n,1) = Missile X position
Rem *** Missiles(n,2) = Missile Y position
```

```
Dim MISSILE(4,3)
MISSILEDELAY=0
```

```
Global SPRX,SPRY,MISSILE(),MISSILEDELAY
```

```
Do
  Bob Clear

  If Joy(1) and 1
    SPRY=SPRY-2
  End If
```

```
If Joy(1) and 2
    SPRY=SPRY+2
End If
If Joy(1) and 4
    SPRX=SPRX-2
End If
If Joy(1) and 8
    SPRX=SPRX+2
End If
```

```
Rem *** Has firebutton been pressed?
If Joy(1) and 16
    _FIREMISSILE
End If
```

```
Rem *** Update and redraw all missiles
    _MOVEMISSILES
```

```
Rem *** Redraw ship bob...
Bob 1,SPRX,SPRY,1
```

```
Bob Draw : Sprite Update
Screen Swap 0
Wait Vbl
```

Loop

```
Procedure _FIREMISSILE
```

```
    MISSILENUM=-1
```

```
    MXDELAY=14 : Rem *** Delay between firing of missiles
```

```
Rem *** Has enough time passed before firing next missile?
If MISSILEDELAY>MXDELAY
```

```
    Rem *** Is there a missile available?
```

```
    For C=0 To 3
```

```
        If MISSILE(C,0)=0
```

```
            MISSILENUM=C
```

```
        End If
    Next C

    Rem *** Has a spare missile been found?
    If MISSILENUM>-1

        Rem *** Initialise missile...
        MISSILE(MISSILENUM,0)=1
        MISSILE(MISSILENUM,1)=X Hard(SPRX)+22
        MISSILE(MISSILENUM,2)=Y Hard(SPRY)+7

        MISSILEDELAY=0

        Rem *** Play fire sound effect...
        Sam Play 1
    End If
End If
End Proc

Procedure _MOVEMISSILES
    SPEED=10 : Rem *** Speed of missiles...

    For C=0 To 3

        Rem *** Has missile been fired?
        If MISSILE(C,0)=1
            Rem *** Draw sprite...
            Sprite 2*(C+10),MISSILE(C,1),MISSILE(C,2),3

            Rem *** Move missile...
            MISSILE(C,1)=MISSILE(C,1)+SPEED

            If MISSILE(C,1)>X Hard(320)
                MISSILE(C,0)=0
                Sprite Off 2*(C+10)
            End If
        End If
    End If
```

Next C

```
Rem *** Increase missile delay counter...
```

```
MISSILEDELAY=MISSILEDELAY+1
```

```
End Proc
```

---

## Keeping up the speed

No one could possibly cast doubts on AMOS's admirable turn of speed, but things can slow down occasionally when using blitter objects. The problem isn't so bad with hardware sprites, simply because they are not drawn as part of the screen bitmap. But if you've ever written a game or demo that uses more than a couple of highly colourful bobs, then chances are you've already encountered this rather annoying problem – although AMOS keeps everything running smoothly, the redraw rate drops to such a slow pace that even displays that are sync'ed correctly become very jerky indeed.

One of the best ways to increase the speed of bob redraws is the decrease the number of colours you use on your game screens – as any games programmer will tell you, the more colours you use, the slower your screen updates. Most arcade games will squeeze into an eight colour screen, and don't forget that you can increase the number of apparent colours by making use of hardware sprites (they use colour registers 17 through to 31) and even the 'Rainbow' command we covered in the last chapter.



### Use of colours

Another reason why bobs can be very slow is caused by AMOS' insistence on taking most of the work away from the programmer. AMOS actually shields the programmer from most of the drudgery usually associated with screen, sprite and bob handling. Take dual playfield displays, for example. To an assembler programmer, a dual playfield display is simply two separate bitmaps which he must manually keep track of. Whenever he needs to swap the two screens, this must be performed manually by redirecting screen DMA to the logical screen. AMOS, on the other hand, does all this work for you – just issue the 'Double Buffer' command and AMOS will then automatically swap the physical and logical screens every 50th of a second.

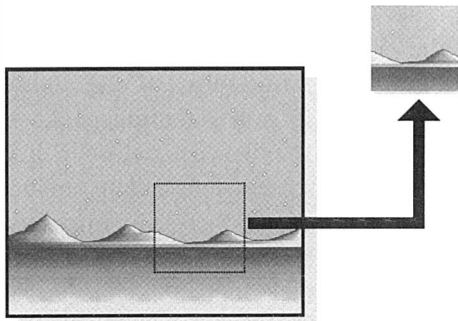
You're probably wondering why I am criticising what is undoubtedly one of AMOS' strengths. After all, the easier it is to program a computer, the better, surely? To an extent this is true, but unfortunately AMOS's simplicity comes at a price. Because AMOS's bob handling has not been optimised for a particular program, there's an unavoidable loss in speed. This decrease in speed isn't always noticeable, but I can guarantee you'll notice it if you try writing a game!

As always, though, AMOS offers us a solution. Using a couple of very powerful commands built into every version of AMOS (yes, even Easy AMOS), it is possible to speed up most programs by turning off AMOS's automatic bob handling. This obviously creates more work for the programmer (after all, if AMOS isn't looking after your sprites and bobs, it's down to you), but the little bit of extra work involved is more than worth the effort when compared to the increase in the rate of screen redraw. Programs that veritably crawled along using AMOS' automatic screen and bob handling can be totally transformed into the sort of super slick affairs that we've all come to expect from AMOS.

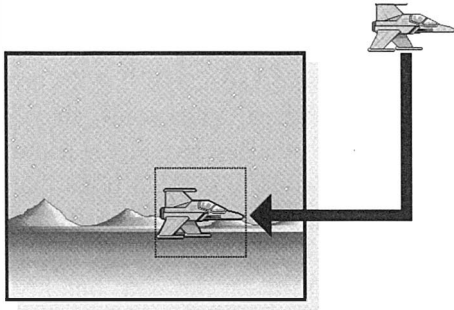


### Bob handling

Because AMOS takes away so much of the work involved in moving blitter objects, few of us truly appreciate what goes on whenever the position of a bob is changed. Unlike sprites (don't forget that they aren't technically part of the screen display), the Amiga's blitter has to carry out some pretty complex operations in order to move any bob. To appreciate this process more fully, let's take a look at exactly what's involved:

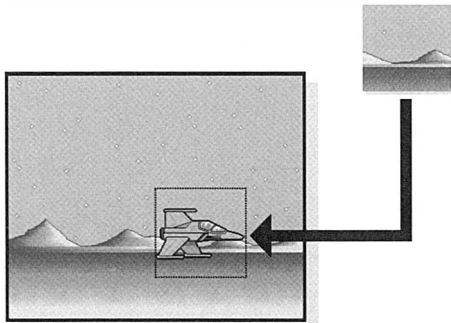


**I** When your program tells AMOS to display a bob, the first thing that the blitter does is to transfer a rectangular area of the screen that will be obscured by the Bob to a temporary 'safe' area of memory.



**2** With the background image safely stored, the Bob is pasted down directly into the screen bitmap, therefore destroying the background graphics beneath it.

**3** Wait for a vertical blank before proceeding...



**4** The bob is now removed by copying the rectangular area of background graphics that was stored in a temporary memory location back to the screen bitmap.

**5** With the entire bob redrawing process complete, loop back to step 1.

As you can see, moving a bob isn't simply a process of redrawing it at a new screen position. Most of the time, all this work is carried out almost instantly by the very rapid blitter chip inside your Amiga. But things can start to slow down considerably when you attempt to move more than one blitter object at once. Because the blitter is forced to redraw all your bobs one at a time before the screen can be swapped, the amount of setting up required to make the blitter work causes an unavoidable slowing down.

In order to keep things running as fast as possible, though, AMOS allows us to turn off the automatic redrawing of blitter objects that causes the

reduction in speed. All you need to do is to issue the ‘Bob Update Off’ command at the start of your program and AMOS will no longer redraw blitter objects unless you expressly tell it to do so. Even if your program directly moves blitter objects using the ‘Bob’ command, they won’t actually be drawn into the logical screen (the hidden screen) until you issue the ‘Bob Redraw’ command. What’s more, the old bobs that have to be removed won’t be removed until you issue the ‘Bob Clear’ command. Let’s take a look at some skeleton code that makes this process somewhat clearer (you won’t find this in the disk bundled with this book):

```
Rem ** Start of Program **
Bob Update Off

Do
  Rem *** Remove all our old bobs
  Bob Clear

  Rem ** Move Bobs now **

  Rem *** Draw all bobs onto screen
  Bob Draw
  WaitVBL
Loop
```



Before you type in this code only to find that it doesn’t work (hence the reason I haven’t put it on the disk!), I must stress that it is purely skeleton code. That is, it serves only to demonstrate the placement of the three Bob commands discussed above. As an example of how to use controlled bob redraws, though, it serves our purposes beautifully.

As you can see from the listing, the ‘Bob Update Off’ command is issued at the very start of our program. This only needs to be executed once, so there’s no need to place it inside the program’s main loop. It really acts as nothing more than a toggle that turns AMOS’s automatic bob updating facility on and off. You can therefore (if you really want to) turn bob updating back on again using the ‘Bob Update On’ command.

Moving on, the listing then enters a loop that forms the backbone of all well programmed computer games. The very first instruction that should be performed within your main game loop is the ‘Bob Clear’ command that – not surprisingly – removes all bobs that have been previously drawn into the logical screen by redrawing the background graphics that they obstructed. Once this is done, the rest of the loop is performed as normal without any changes needing to be made.

Even if the loop moves the bobs, they will not be drawn onto the screen. Instead they will all be buffered up ready to be performed as soon as you give the go ahead. It’s a bit like turning a stop valve on and off – if you leave the stop valve open, you’ll get a very slow trickle of water. But close it for a while and then re-open it and you’ll get a sudden rush of water – although the same amount of water has passed through the stop valve, restraining its flow until you were ready allowed a greater amount of water to pass through in less time. Think of the water as blitter operations and you won’t go far wrong!

The instruction that opens that stop valve is none other than ‘Bob Draw’, which instructs AMOS to perform all bob movements that were requested since the last time the instruction was called. And, because your bobs are being moved ‘en masse’, AMOS will move them considerably faster than would have previously been thought possible. Finally, the whole bob drawing process is sync’ed in with the vertical blanking period using the ‘WaitVbl’ instruction before the screen is automatically brought into view.

If your bobs are being drawn onto a plain background, then it’s worth taking advantage of the ‘Set Bob’ command’s ability to turn off the automatic screen redrawing process normally performed whenever a bob is moved. As you will know, when you place a bob on top of a graphic, the area of the graphic is saved into memory. When you then move the bob to another part of the screen, the background is restored automatically by transferring the area of saved background graphics back to the screen.



Whilst this is necessary if your bobs are moving over a picture (or whatever), it isn’t needed if the background is plain. You must, however,



tell AMOS what colour the background is. For example, if the background was coloured using colour register 3, you would use the following command:

---

```
Set Bob BOBNUMBER,4,,
```

 AMOS  
COMMAND DEFINITION

Note how a value of 4 is passed instead of the actual colour register number 3. Don't worry too much about this – it's just a peculiarity of AMOS. Just bear in mind that you must add 1 to the colour that you pass to the 'Set Bob' command.



Remember screen synchronisation, which we covered in the last chapter? Screen synchronisation plays a very important role in bob handling too. Because bobs are drawn into a screen bitmap just like any other graphic (our scrolling routines, for example), it's equally important to keep AMOS's double buffering on the leash when using bobs too. If you want to keep your bobs moving smoothly, I strongly suggest you turn off AMOS's automatic screen flipping facility (using AutoBack 0) and swap screens manually using the 'Screen Swap' instruction.

Switching to AutoBack 0 mode does have its disadvantages, however. Because the entire process of updating a double buffered display is left to the programmer, drawing normal graphics directly onto the display is complicated somewhat.

Say, for example, you wanted to write some text onto the screen at a given screen position. If you were running under AutoBack 0, it would be left down to you to ensure that both the physical and logical screens are updated. AMOS does come to the rescue here a bit, however, with AutoBack mode 1. Simply change 'AutoBack 0' to 'AutoBack 1' and AMOS will still allow you complete control over screen flipping, but will continue to make sure that graphics (not including Bobs) are drawn into both the physical and logical screens.

Here's a quick demonstration program that shows how much smoother bob movement is when handled manually:



```
Rem *** Smooth Bob Movement Demonstration
Rem *** Filename - BobUpdate.AMOS
```

```
Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0
```

```
Screen Open 1,320,17,2,Lowres
Flash Off : Curs Off : Cls 0
Palette $F00,$FFF
```

```
Locate 0,0 : Centre "Move ship with joystick and then"
Locate 0,1 : Centre "Press SPACE BAR to make Bob smooth"
```

```
Screen 0
Load "AMOSBOOK:ABKFiles/LargeShip.ABK"
Get Sprite Palette
Double Buffer
```

```
SPRX=0 : SPRY=128 : FRAME=1 : FRAMEDELAY=0
Global SPRX,SPRY,FRAME,FRAMEDELAY
```

```
Rem *** Normal bob update...
Repeat
    _MOVESHIP
    Bob 1,SPRX,SPRY,FRAME
    Wait Vbl
Until Inkey$=" "
```

```
Screen 1 : Cls 0
Centre "Now for some smooth movement..."
```

```
Screen 0
Bob Off 1
Cls 0
Set Bob 1,1,,
Bob Update Off
Autoback 0
```

```
Rem *** Manual Bob update with Autoback 0...
```

```
Do
```

```
    Bob Clear
```

```
    _MOVESHIP
```

```
    Bob 1, SPRX, SPRY, FRAME
```

```
    Bob Draw
```

```
    Screen Swap 0
```

```
    Wait Vbl
```

```
Loop
```

```
Procedure _MOVESHIP
```

```
    If Joy(1) and 1
```

```
        SPRY=SPRY-4
```

```
        If SPRY<-40
```

```
            SPRY=-40
```

```
        End If
```

```
    End If
```

```
    If Joy(1) and 2
```

```
        SPRY=SPRY+4
```

```
        If SPRY>230
```

```
            SPRY=230
```

```
        End If
```

```
    End If
```

```
    If Joy(1) and 4
```

```
        SPRX=SPRX-4
```

```
        If SPRX<-180
```

```
            SPRX=-180
```

```
        End If
```

```
    End If
```

```
    If Joy(1) and 8
```

```
        SPRX=SPRX+4
```

```
        If SPRX>300
```

```
            SPRX=300
```

```
        End If
    End If

    If FRAMEDELAY>2
        FRAME=FRAME+1
        If FRAME=3
            FRAME=1
        End If
        FRAMEDELAY=0
    End If
    FRAMEDELAY=FRAMEDELAY+1
End Proc
```



# AMAL

- Multi-tasking and interrupts
- How AMAL works
- AMAL Editor versus 'embedded' code
- Assigning and handling channels
- AMAL registers – local, global and 'special'
- AMAL instruction set
- AMAL functions
- Using more than 16 channels
- Useful AMAL routines

**M**ost modern computers, including the Amiga, use what can best be described as a ‘serial processor’. These rather ageing devices (even the Amiga 4000/040 is based around a microprocessor design that is older than most AMOS users themselves!) are limited to performing just a single task at once. Even the Amiga’s multitasking capabilities are just a clever form of ‘task switching’ where the main processor shares its time between several programs, performing a few lines from each as it goes. However, the Amiga’s processor runs so fast that you’re unlikely to ever notice the difference between its task switching technique and a true ‘parallel processor’ such as the infamous transputer chip.

Until Commodore get around to designing an Amiga that uses a parallel processor (which is very unlikely anyway), every program you write in AMOS – and indeed any programming language – is executed by the Amiga an instruction at a time. Obviously this doesn’t impose too many limitations if you’re writing a fairly simple utility or demo, but games are considerably more complex. Every fiftieth of a second an average game will perform a myriad of different tasks in order to keep everything running smoothly. Even on an Amiga 4000, you may often find that your program is so complex that the Amiga is unable to update the game fast enough to keep it running at an acceptable rate. The AMOS compiler can help, but even compiled games may still not be fast enough.

Thankfully Motorola, the company that produces the processor inside your Amiga, took this into account and they very cleverly built into every 68000 series processor what are known as ‘interrupts’. Most processors offer interrupts in one form or another, but the Amiga’s processor is particularly well served. An interrupt is simply a clever hardware-level trick that forces the Amiga’s processor to run several programs simultaneously using a task-switching technique similar to that used by the Amiga’s multitasking operating system. Every 50th of a second or so, the Amiga switches from the program that it is running and runs another separate program instead. Both the program and any interrupt-driven programs that you create are assigned their own ‘time slice’, effectively giving the Amiga the ability to run programs in parallel.



### Interrupts

AMOS supports interrupt-driven routines too in the form of AMAL, a very powerful facility that is effectively a separate programming language in its own right. AMAL (which, for the nosy amongst you, stands for ‘AMOS Animation Language’) has been specifically designed to handle the sort of mundane tasks that you would normally have been forced to handle yourself, therefore eating up valuable processor time. Obviously, AMAL programs eat up processor time themselves, but the amount is miniscule when compared to normal AMOS code. Because each and every AMAL instruction works at a very low-level, they run considerably faster than their AMOS counterparts, making AMAL a valuable time saver.

As its full name suggests, AMAL is primarily designed for handling the movement and animation of objects. As a result, it gives us a nice and easy (and very fast!) method of producing the sort of smooth animation effects usually associated with commercial arcade games. AMAL is capable of handling three different types of object – sprites, bobs and screens (yes, AMAL treats screens as just another type of object that can be moved around your monitor or TV display). The great thing about AMAL (apart from the obvious time savings) is the fact that once you’ve set an AMAL program running, you can virtually forget about it and carry on with the rest of your program. If you’ve written an AMAL ‘program’ to move and animate a Bob, for example, it will continue to fly around the screen even if your program is doing something entirely different!



Easy AMOS

Before we go any further, I’m afraid I’ve got some bad news for Easy AMOS users – Easy AMOS doesn’t support AMAL in any shape or form. As a result, unless you’re considering upgrading to either AMOS 1.35 or – better still – AMOS Professional, I’m afraid this entire chapter is going to be of little use to you. Sorry guys!

---

## AMAL principles

In order to get AMAL to do anything even remotely useful, you need to write an AMAL ‘program’. AMAL programs can be created in one of two ways – either using the ‘AMAL Editor’ accessory bundled with AMOS or by embedding them directly into your program code. Most



AMAL programs can be coded directly into a memory bank using the AMAL Editor bundled with AMOS. Although this makes AMAL coding faster, your AMAL programs will not be visible from the main AMOS Editor.

```

AMAL string editor      C: 0      L: 0      AMAL channel number 0
E00000000000111111
E0123456789012345
A: Let R1 = 0
   Let R1 = R1 + 4
   If R1 > 320 Jump B
   Jump C
B: Let R1 = 0
   Jump C
C: Let X = R1
   Pause
   Jump A

```

programmers prefer the latter approach because it allows you to edit your AMAL programs alongside the AMOS program that will use them. The AMAL Editor, on the other hand, stores AMAL programs into a memory bank. Although this gives you a far greater level of code security (your AMAL programs will be invisible to anyone who takes a sneaky look at your source code!), it can make program development somewhat more complex as you will have to load the AMAL Editor each time you wish to make a change to your AMAL programs. For these reasons, we shall not be covering the AMAL Editor within this book. If you want to find out more, then check your user's guide.



**AMAL Editor**



**'Embedding'  
AMAL code**

Embedding an AMAL program into an AMOS program is very easy indeed. All you have to do is to assign the AMAL program to a string variable at the start of your program. Although the AMOS Editor can horizontally display only 80 characters at once, you can assign enormous AMAL programs to the same string variable using something like the following:

```

A$ = "Your AMAL program code goes here. If you want"
A$ = A$ + "to add extra code then you'll have to append"
A$ = A$ + "the extra lines using this technique."

```



Once you've defined your AMAL program, you need to do a bit of setting up to get it running. Just like the 'Anim' command we covered in chapter 7, AMAL programs are run under interrupt using what are

known as ‘channels’. Before you can assign your AMAL program to a channel, however, it’s important that you start by telling AMAL which object you wish the AMAL program to affect.

By default, AMAL automatically assigns the first 16 channels to the first 16 hardware sprites, but it’s far better programming practice to specifically tell AMAL which object you wish it to use. For sprites and bobs, AMAL offers two strains of the ‘Channel’ command:

---

```
Channel CHANNEL To Sprite SPRNUM
Channel CHANNEL To Bob BOBNUM
```



**CHANNEL** The ‘Channel’ parameter is a value between 1 and 16 that tells AMOS which interrupt channel the object is to be assigned to. Note that in order for this command to work, the object must have been created first using either the ‘Sprite’ or ‘Bob’ commands.

**SPRNUM/BOBNUM** The ‘SprNum’ and ‘BobNum’ parameters are simply the identifier number of the sprite or bob that you wish to tie to the channel.

AMAL isn’t just restricted to controlling sprites and bobs, however. Four more forms of the ‘Channel’ command are also on offer that give you extensive control over the size, position and offset of any screen. If you’ve created a rainbow, AMAL can control this too.

Here are those extra ‘Channel’ commands in all their AMOS glory. The parameter format of them all is fairly obvious – just specify the channel number followed by the identifier number of the screen that you wish to tie to that channel.

---

```
Channel CHANNEL To Screen Display SCRNUM
Channel CHANNEL To Screen Offset SCRNUM
Channel CHANNEL To Screen Size SCRNUM
Channel CHANNEL To Rainbow SCRNUM
```



Once you've assigned an object to an interrupt channel, you're ready to pass your AMAL program to that channel so that it knows what it must do. The command that you need is as follows:

---

Amal CHANNEL, STRING\$



**CHANNEL** The 'Channel' parameter is a value between 1 and 16 that tells AMOS which interrupt channel your AMAL program is to be assigned to. Although you can theoretically create more than 16 channels (as we shall see later in this chapter), only the first 16 channels will run under interrupt.

**STRING\$** The 'String\$' parameter is the name of the string variable that contains your AMAL program. As a result, you should ensure that your AMAL program is defined before calling this command.

With both the object you wish to control and your AMAL program assigned to a interrupt channel, all that remains is to bring the AMAL program to life with the 'Amal On' command. Here's a quick demonstration listing to whet your appetite – not only does it demonstrate how to move and animate a bob under AMAL control, but it even exits the program and goes back to AMOS's 'Direct' mode after everything has been set up, leaving the AMAL program running! Don't worry too much about the AMAL program for the moment – we'll be covering the format of AMAL programs later.

```
Rem *** Simple AMAL Demonstration
Rem *** Filename - AmalDemo.AMOS

Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0

Load "AMOSBOOK:AbkFiles/Helicopter.ABK"
Get Sprite Palette
Double Buffer
```



```
Rem *** Define AMAL Program..

MOVESHIP$=""          Let RA = -48"
MOVESHIP$=MOVESHIP$+" Anim 0, (1,2)(2,2)(3,2)"
MOVESHIP$=MOVESHIP$+"A: Let RA = RA + 4"
MOVESHIP$=MOVESHIP$+" If RA > 320 Jump B"
MOVESHIP$=MOVESHIP$+" Jump C"
MOVESHIP$=MOVESHIP$+"B: Let RA = -48"
MOVESHIP$=MOVESHIP$+"C: Let X = RA"
MOVESHIP$=MOVESHIP$+" Pause"
MOVESHIP$=MOVESHIP$+" Jump A"

Rem *** Create Bob...
Bob 10,0,50,1

Rem *** Assign Channel 1 to our Bob..
Channel 1 To Bob 10

Rem *** Assign AMAL program to Channel 1..
Amal 1,MOVESHIP$

Rem *** Turn AMAL channel on..
Amal On

Rem *** Return to AMOS 'Direct' mode..
Direct
```

---

## AMAL registers

One of the most important fundamentals of just about every programming language is the variable, a sort of electronic pigeon-hole used to store anything from a string of characters to a number. AMOS uses them and AMAL does too, although AMAL's variables are somewhat different. For starters, AMAL calls its variables 'registers' and with good reason too – unlike a 'real' programming language, AMAL does not allow you to give your variables personalised labels. Instead, AMAL provides a set of 36 pre-defined 'registers' which can be used to temporarily store information. What's more, AMAL's variables can only

handle numbers. Even then, any numbers you use must be within the range of 32768 to -32767.



### 'Local' & 'global' registers

AMAL registers come in two flavours – local registers and global registers. Both are very similar except for the fact that global registers are shared between all AMAL programs and local registers are unique to each AMAL program that you run. If you place a value into a global variable, each and every AMAL program that you run will be able to access it. As a result, you should always restrict yourself to using local registers unless you specifically want each and every AMAL program to be able to read and modify the same data. This can be useful if you're using AMAL to scroll a screen using hardware scrolling and you need to keep all your bobs static whilst the screen scrolls – all you have to do is to keep the current screen offset value in a global variable and then add it to the 'X' or 'Y' co-ordinates (depending upon the direction of your scroll).

AMAL provides 26 global registers, each of which starts with the letter 'R' (for 'Register') followed by a letter of the alphabet. The first global variable, for example, is called 'RA' and the last is called 'RZ'. Local variables follow a very similar pattern, but instead of having 26 registers at your disposal, AMOS gives each AMAL program only 10, numbered 0 to 9. Like global registers, however, local registers must start with the letter 'R'. 'R0', for example, is the first local variable and 'R9' is the last.

On the whole, AMAL keeps itself to itself, but there may be times when you need to read and write to AMAL registers from within your main AMOS code. Thankfully, AMOS provides the command to do this too in the shape of 'Amreg()' function. The command has two different formats – one for handling global AMAL registers and one for local registers. First, let's take a look at the format required to read the contents of a global AMAL register:

---

**VALUE** = Amreg(REGISTER)



**VALUE** The variable 'Value' contains the value returned by the 'Amreg()' function. Once the 'Amreg()' function has been called, your AMOS

program will be able to use the data returned simply by reading the contents of this variable.

**REGISTER** The ‘Register’ parameter is a value between 0 and 25 that corresponds to AMAL registers ‘RA’ through to ‘RZ’. A value for ‘0’, for example, would instruct AMOS to return the value of ‘RA’.

---

**VALUE = Amreg(CHANNEL, REGISTER)**



This second strain of the ‘Amreg()’ function allows you to read the contents of an AMAL local variable from any of your AMAL programs that are current running. In order to identify the set of local variables you’re interested in (don’t forget that each and every AMAL program you create can have its own set), you need to specify the number of the interrupt channel that the AMAL program is running under. Finally, the ‘Register’ parameter must contain a number between 0 and 9 that corresponds to AMAL registers ‘R0’ to ‘R9’.

You can also write to an AMAL register simply by reversing the format of either form of the ‘Amreg()’ command. Say, for example, you wanted to write a value held in a variable called ‘MYNUM’ to the local register ‘R3’ for an AMAL program running under interrupt channel ‘4’. All you’d need is the following command:

**Amreg(4,3) = MYNUM**



AMAL also provides three special AMAL registers that are ‘hard-wired’ into the objects that your AMAL programs can control. These are:

**X** This special register holds the current X co-ordinate of the object your AMAL program is controlling. If you’re controlling a bob, then the co-ordinate will be in the form of a screen co-ordinate. For screens and hardware sprites, though, the co-ordinate will be a hardware co-ordinate.

In the case of the ‘Rainbow’ form of the ‘Channel’ command, the ‘X’ register holds the value of the first colour in the rainbow. By changing this colour, the rainbow will appear to cycle.

**Y** Not surprisingly, this register holds the current Y co-ordinate of the object current under AMAL control. Once again, the same co-ordinate formats apply.

The ‘Y’ register has a slightly different purpose when AMAL is used to control a screen’s size and a rainbow. In the case of screen resizing, the ‘Y’ register holds the current height of the screen. By changing this value, you can shrink and expand a screen under AMAL control. For Rainbows, the ‘Y’ register will contain the line on the screen (as a hardware co-ordinate) where the rainbow effect starts.

**A** The ‘A’ register holds the number of the current image assigned to the object under AMAL control. If the object is being animated, the contents of the ‘A’ register will change accordingly.

When used within an AMAL program that controls a screen’s size, the ‘A’ register holds the screen’s current width. For rainbows too, the ‘A’ register has a different role – it holds the height of the rainbow in scan lines. By changing this value, you can easily shrink and expand a rainbow under interrupt.

---

## The AMAL instruction set

When compared to AMOS, the AMAL instruction set is surprisingly small. However, you’ll be surprised just how capable this small selection of AMAL commands are. Before we go any further, however, it’s worth saying a few words about the format of AMAL programs. Unlike AMOS, AMAL is very touchy about case sensitivity. Although all the commands documented below are listed using their full names, to be perfectly honest, AMAL only takes notice of the first one or two characters. Even then, these special characters must be entered as capitals. Anything that you enter in lower case will be ignored completely by AMAL. Take the AMAL command ‘Move’, for example. The only letter that AMAL actually understands is ‘M’ – the ‘ove’ bit is entered simply to make AMAL programs more readable. In theory, there’s nothing stopping you from entering ‘M’, ‘Move’, ‘Movement’ or even ‘Mildred’.



### Capital letters

Right, now that we've got that out of the way, let's get stuck into AMAL's instruction set:

---

**Move DELTAX, DELTAY, NUMBEROFSTEPS**

The 'Move' command ('M') moves an object using a set of 'delta' values defined by the two parameters 'DELTAX' and 'DELTAY'. If you've never encountered delta values before, they're simple enough. Instead of specifying an object's new position in relation to the top left-hand corner of the screen (the conventional method), the values are entered relative to the object's last position. If, for example, you entered a 'DELTAX' value of '2' and the object you were trying to move was at an X-position of '100', it would move to position '102'.

The 'NumberOfSteps' parameter tells AMAL how many vertical blanks it should spread the movement across. For example, if you wanted to move an object 100 pixels across the screen over 5 vertical blanks, the object would move at a rate of 20 pixels per vertical blank.

---

**Anim REPEAT, (IMAGE, DELAY), (IMAGE, DELAY)...**

The 'Anim' command ('A') works in virtually the same way as AMOS's own 'Anim' command which we covered in Chapter 7. It can be used to animate an object by changing the image attached to the object from the sprite bank. Each frame of your animation is enclosed between brackets with two parameters – the image number ('IMAGE') and the length of time in vertical blanks that the image is to be displayed for ('DELAY'). Any number of frames can be defined and you can easily mix and match images to your heart's content.

The 'Repeat' parameter tells AMAL how many times the animation is to be repeated. If you specify a value of '0', the animation will loop indefinitely.



---

**Jump LABEL**

The AMAL 'Jump' command ('J') is identical to AMOS's own 'Goto' command and is used to branch from one part of an AMAL program to another. The 'LABEL' parameter is a legal AMAL label. AMAL allows you to define labels using any of the characters in the alphabet providing that each label you define ends with a colon (':'). Once again, only the first character (which must be a capital) is recognised.

---

**Let REGISTER = EXPRESSION**

The 'Let' command ('L') assigns a value to a register. The 'Expression' parameter can be just about any legal AMOS expression. For example, you could assign the contents of the AMAL register 'R1' to 'R2' and automatically add a value of '4' to it using 'Let R2 = R1 + 4'.

---

**If TEST Jump LABEL**

The AMAL 'If...Jump' command ('I' and 'J') allows you to perform a test (is the value in register 'R0' greater than 10, for example?) and then branch to another part of your AMAL program if the result is true.

---

**For REGISTER = START To END... Next REGISTER**

Pretty self-explanatory this one. If you're used to using AMOS's own 'For...Next' construct, then you'll instantly recognise this command. It simply performs a loop by incrementing the contents of 'Register' (which is an AMAL register) using a set of start and end values defined by the parameters 'Start' and 'End'. The Loop is then closed by calling the 'Next' ('N') command.

---

**Pause**

---

The ‘Pause’ command (‘P’) is AMAL’s version of the AMOS ‘Wait Vbl’ command. It is used to temporarily halt the execution of an AMAL program until the next vertical blank occurs. In order to keep your AMOS and AMAL programs running in sync, this command should always be used.

---

---

**AMAL functions**

Although AMAL’s commands form the backbone of an AMAL program, the real workhorses of any AMAL program are AMAL’s impressive set of functions, which allow you to monitor the activity of a range of different events. You can, for example, read the status of a joystick, check for collisions between objects and even convert hardware co-ordinates to screen co-ordinates and vice-versa. Let’s take a look at each in turn.

---

**VALUE = Bob Col(BOBNUMBER, FIRSTBOB, LASTBOB)**  
**VALUE = Sprite Col(SPRNUMBER, FIRSTSPR, LASTSPR)**



---

AMAL’s ‘Bob Col()’ function (‘BC()’) is identical to AMOS’s own ‘Bob Col’ command. It’s used to check for collisions between a specified bob (‘Bobnumber’) and a range of other bobs between ‘FirstBob’ and ‘LastBob’. If a collision is detected, a value of ‘-1’ is returned. It’s worth noting, however, that this function does not work unless your AMAL program is run directly with the AMOS ‘Synchro’ command. We’ll be covering this next.

If the ‘BC’ function checks for collisions between blitter objects, then I’m sure you’ll already have guessed that the ‘Sprite Col()’ function (‘SC()’) does the same job for hardware sprites. Once again, however, this function will not work unless your AMAL programs are run directly with the ‘Synchro’ command. Note also that you must still create a mask for each hardware sprite that you wish to check, using the ‘Make Mask’ command covered in Chapter 7.

---

**VALUE = Col (OBJECTNUMBER)**



This function ('C()') returns the status of a specified object ('Objnumber') after a collision check. If the object was involved in a collision, a value of '-1' will be returned.

---

**VALUE = Joy0**

**VALUE = Joy1**



If you need to read the status of joystick port 2 from within your AMAL programs, then you need this function. 'Joy0' ('J0') returns the status of the joystick in the form of a bitmap. Not surprisingly, the 'Joy1' ('J1') function returns the status of control port 1 (the mouse) port whenever a joystick is connected to that port. Once again, the status of the joystick is returned in the form of a bitmap. Refer back to Chapter 8 for more information on the format of this bitmap.

---

**VALUE = Key1**

**VALUE = Key2**



The 'Key1' function ('K1') returns the status of the left mouse button. If it is being pressed, a value of '-1' will be returned. Otherwise, a value of '0' is returned. The 'Key2' function ('K2'), on the other hand, returns the status of the right mouse button. The return values are exactly the same – '0' if the button isn't being pressed and '-1' if it is.

---

**VALUE = VU (VOICE)**



If you're writing a demo, then you'll almost certainly want to include what demo coders call 'VU meters', that is, those flashy bouncing bar effects that look just like the 'VU' meters on a real stereo. The 'VU()' function returns a value between 0 (silent) and 63 (loud) for the current intensity of a sound voice specified with the 'Voice' parameter.

---

**XHARDCOORD = XHard (SCREEN, SCRXC OORD)**  
**YHARDCOORD = YHard (SCREEN, SCRYC OORD)**




---

If you need to be able to convert hardware and software co-ordinates from within your AMAL programs, then help is at hand. AMAL includes all four of the functions present in AMOS's own instruction set. The first of these, 'XHard()' ('XH()') converts a screen 'X' co-ordinate held in the 'SCRXC OORD' parameter to a hardware co-ordinate relative to the screen defined by the 'Screen' parameter. The 'YHard()' function ('YH()') converts a screen 'Y' co-ordinate to a hardware 'Y' co-ordinate relative to the specified screen.

---

**XSCRCOORD = XScreen (SCREEN, HARDXC OORD)**  
**YSCRCOORD = YScreen (SCREEN, HARDYC OORD)**




---

AMAL allows you to convert hardware co-ordinates to screen co-ordinates too using this pair of functions. The 'XScreen()' function ('XS()') converts a hardware 'X' co-ordinate to a screen 'X' co-ordinate and the 'YScreen()' function ('YS()') converts a hardware 'Y' co-ordinate to a screen 'Y' co-ordinate. Once again, the values are taken relative to a screen defined by the 'Screen' parameter.

---

**XHARDCOORD = XMouse**  
**YHARDCOORD = YMouse**




---

These two functions are pretty obvious – the 'XMouse' function ('XM') returns the 'X' co-ordinate of the mouse pointer and the 'YMouse' function ('YM') returns the 'Y' co-ordinate of the mouse pointer. Once again, the co-ordinates are returned in the form of hardware co-ordinates.

---

**VALUE = Z (NUMBER)**




---

If you need to generate a random number from within your AMAL program, then this is the function for you. Simply by passing it a 'seed'

value in the 'Number' parameter, the 'Z()' function returns a random number. To be perfectly honest, the 'Z()' function is not the greatest random number generator – if you need to create truly random values, then it may be best use AMOS's own 'Rnd()' function and pass the values to your AMAL programs using the 'Amreg()' function.

---

## Beyond 16 channels...



**SYNCHRO**  
command

Under normal circumstances, AMAL allows you to run no more than 16 AMAL channels at once. This is caused by the Amiga's processor being unable to handle any more than this number. If you need more channels, however, you can get around this limitation using the 'Synchro' command. The 'Synchro' command allows you to turn off the Amiga's hardware-based interrupts and run the programs directly from your AMOS program. Obviously, this has the unavoidable side effect of stopping programs running under true interrupts, but it's unlikely that you'll notice the difference.



**Collision**  
detection

One big benefit of running AMAL programs directly is that not only are you no longer restricted to just 16 AMAL channels (you can create up to 64 AMAL channels with the Amiga's interrupts turned off!), you can also make use of AMAL's collision detection functions 'BC()', 'SC()' and 'C()'. When running AMAL programs directly, it's perfectly possible to write games and demos that run entirely under AMAL!

In order to make use of this facility, you need to start your program by turning off the Amiga's hardware interrupts using the 'Synchro Off' command. Once this is done, you can define your AMAL programs and then run them. However, each time any of your AMAL programs waits for a vertical blank (using the AMAL 'Pause' function), you have to specifically tell them when to start running again. If you don't, they'll just sit there dumbly.

The command to start an AMAL program running again after a vertical blank is 'Synchro'. This tells all AMAL channels that are currently halted to start running again. Here's a quick demonstration:



```
Rem *** 'Synchro' Demonstration
Rem *** By Jason Holborn
```

```
Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0
```

```
Load "AMOSBOOK:AbkFiles/Helicopter.ABK"
Get Sprite Palette
Double Buffer
```

```
Rem *** Define two AMAL programs...
```

```
MOVESHIP$=""           Let R1 = -48"
MOVESHIP$=MOVESHIP$+"  Anim 0, (1,2) (2,2) (3,2)"
MOVESHIP$=MOVESHIP$+"A: Let R1 = R1 + 4"
MOVESHIP$=MOVESHIP$+"  If R1 > 320 Jump B"
MOVESHIP$=MOVESHIP$+"  Jump C"
MOVESHIP$=MOVESHIP$+"B: Let R1 = -48"
MOVESHIP$=MOVESHIP$+"C: Let X = R1"
MOVESHIP$=MOVESHIP$+"  Pause"
MOVESHIP$=MOVESHIP$+"  Jump A"
```

```
MOVESHIP2$=""          Let R1 = 320"
MOVESHIP2$=MOVESHIP2$+" Anim 0, (1,2) (2,2) (3,2)"
MOVESHIP2$=MOVESHIP2$+"A: Let R1 = R1 - 4"
MOVESHIP2$=MOVESHIP2$+"  If R1 < -48 Jump B"
MOVESHIP2$=MOVESHIP2$+"  Jump C"
MOVESHIP2$=MOVESHIP2$+"B: Let R1 = 320"
MOVESHIP2$=MOVESHIP2$+"C: Let X = R1"
MOVESHIP2$=MOVESHIP2$+"  Pause"
MOVESHIP2$=MOVESHIP2$+"  Jump A"
```

```
Rem *** Turn off AMAL interrupts...
Synchro Off
```

```
Rem *** Create five Bobs...
```

```
Bob 1,0,20,1
Bob 2,0,60,1
Bob 3,0,100,1
Bob 4,0,140,1
Bob 5,0,180,1

Rem *** Assign Channels to our Bobs...
Channel 1 To Bob 1
Channel 2 To Bob 2
Channel 3 To Bob 3
Channel 4 To Bob 4
Channel 5 To Bob 5

Rem *** Assign AMAL program to all five Channels...
Amal 1,MOVESHIP$
Amal 2,MOVESHIP2$
Amal 3,MOVESHIP$
Amal 4,MOVESHIP2$
Amal 5,MOVESHIP$

Rem *** Turn on all AMAL channels...
Amal On

Repeat
    Rem *** Run AMAL programs again...
    Synchro

    Wait Vbl
Until Mouse Key
```

---

## Useful AMAL routines

So far we've covered an awful lot of theory, but we haven't really applied AMAL to the sort of real programming tasks you'd commonly find in a game. Over the next few pages or so, however, you'll find a variety of listings that cover the more common uses of AMAL.

## Moving a bob using a joystick

The first AMAL routine demonstrates how to move a bob around the screen with the joystick completely under AMAL control. There's nothing spectacular about this routine – it simply checks the status of the joystick and updates a pair of co-ordinates held in the 'R1' and 'R2' registers before passing them to the 'X' and 'Y' registers tied to the bob in question.

```

Rem *** AMAL Joystick Bob control
Rem *** Filename - AMALJoystick.AMOS

Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0
Double Buffer

Load "AMOSBOOK:AbkFiles/Helicopter.ABK"
Get Sprite Palette

Rem *** Set Bob position to default values...
A$="      Let R1=160"
A$=A$+"  Let R2=128"
A$=A$+"  Anim 0, (1,2) (2,2) (3,2)"

Rem *** Check status of joystick...
A$=A$+"A:  If Joy1 & 1 Jump B"
A$=A$+"   If Joy1 & 2 Jump C"
A$=A$+"   If Joy1 & 4 Jump D"
A$=A$+"   If Joy1 & 8 Jump E"
A$=A$+"   Jump J"

Rem *** Update and check 'Y' co-ordinate...
A$=A$+"B:  Let R2 = R2 - 4"
A$=A$+"   If R2 < 0 Jump F"
A$=A$+"   Jump J"
A$=A$+"C:  Let R2 = R2 + 4"
A$=A$+"   If R2 > 235 Jump G"
A$=A$+"   Jump J"

```





```
Rem *** Update and check 'X' co-ordinate...
```

```
A$=A$+"D:  Let R1 = R1 - 4"  
A$=A$+"    If R1 < 0 Jump H"  
A$=A$+"    Jump J"  
A$=A$+"E:  Let R1 = R1 + 4"  
A$=A$+"    If R1 > 272 Jump I"  
A$=A$+"    Jump J"
```

```
Rem *** Reset 'Y' co-ordinate...
```

```
A$=A$+"F:  Let R2 = 0"  
A$=A$+"    Jump J"  
A$=A$+"G:  Let R2 = 235"  
A$=A$+"    Jump J"
```

```
Rem *** Reset 'X' co-ordinate...
```

```
A$=A$+"H:  Let R1 = 0"  
A$=A$+"    Jump J"  
A$=A$+"I:  Let R1 = 272"  
A$=A$+"    Jump J"
```

```
Rem *** Pass co-ordinates to bob...
```

```
A$=A$+"J:  Let X = R1"  
A$=A$+"    Let Y = R2"  
A$=A$+"    Pause"  
A$=A$+"    Jump A"
```

```
Bob 1,160,128,1
```

```
Channel 1 To Bob 1
```

```
Amal 1,A$
```

```
Amal On
```

```
Direct
```

## Hardware scrolling a screen

AMAL can be used to great effect in handling the simple task of hardware scrolling a screen, too. All you do is open your screen and then assign the screen to the 'Screen Offset' form of the 'Channel' command. The screen can then be scrolled under AMAL control by writing different values to the AMAL 'X' and 'Y' registers. Don't forget that in order to 'wrap' the screen around, your AMAL program must continuously check for the screen offset values rising or falling above or below a set of minimum and maximum values. Here's a demonstration listing that shows how it's done:

```

Rem *** AMAL Hardware Scroll demonstration
Rem *** Filename - AMALHardScroll.AMOS

Screen Open 0,640,256,16,Lowres
Flash Off : Curs Off : Cls 0
Screen Display 0,128,48,320,256

Load Iff "AMOSBOOK:Pictures/AMALBackground.IFF"
Screen Copy 0,0,0,320,256 To 0,320,0

Rem *** Set initial value of screen...
A$=""      Let RA = 0"

Rem *** Add 2 to screen offset...
A$=A$+"A:  Let RA = RA + 6"

Rem *** Check that screen offset has not exceeded 320...
A$=A$+"    If RA > 320 Jump B"
A$=A$+"    Jump C"

Rem *** Reset screen offset if it has...
A$=A$+"B:  Let RA = 0"
A$=A$+"    Jump C"

Rem *** Pass 'RA' to screen offset 'X' co-ordinate...
A$=A$+"C:  Let X = RA"

```



```
A$=A$+"    Pause"
A$=A$+"    Jump A"
```

Channel 1 To Screen Offset 0

```
Amal 1,A$
Amal On
```

Direct

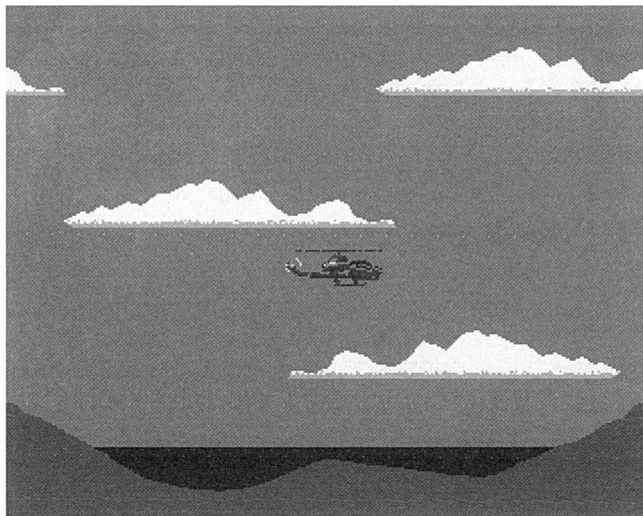
### Keeping one AMAL program in sync with another

Global variables come in very handy when you need to keep one AMAL program in sync with another. Take, for example, the task of moving a Bob over a background that is being hardware scrolled. If you were to simply move the Bob with no consideration for the current 'Screen Offset' values, the Bob would soon disappear off of the screen. By writing the screen's current 'X' and 'Y' positions into a set of global variables, however, you can keep your Bobs on-screen simply by adding the screen offset values to the Bob's 'X' and 'Y' co-ordinates. In the case of the listing below, the screen's 'X' offset is written into the global variable 'RA', which is automatically added to the 'X' co-ordinate of the Bob each time the AMAL 'X' register is updated.

*AMAL can easily handle both hardware scrolling and the positioning of blitter objects, leaving you to concentrate on more important aspects of your game's design.*



**Using AMAL  
for scrolling &  
bobs**



This example program demonstrates how much of a game's code can be handed over to AMAL, letting you concentrate on the gameplay. Note too how this program is run directly with the 'Synchro' command – as your AMAL programs become more complex, you'll find that this can often give smoother results – especially if the program is to be compiled.

```

Rem *** AMAL Bob + Scroll demonstration
Rem *** Filename - AMALBobScroll.AMOS

Screen Open 0,640,256,16,Lowres
Flash Off : Curs Off : Cls 0
Screen Display 0,128,48,320,256

Load "AMOSBOOK:AbkFiles/Helicopter.ABK"
Get Sprite Palette

Load Iff "AMOSBOOK:Pictures/AMALBackground.IFF"
Screen Copy 0,0,0,320,256 To 0,320,0
Double Buffer : Autoback 1
Bob Update Off

Rem *** AMAL Bob Control program...

Rem *** Set Bob position to default values...
A$="      Let R1=160"
A$=A$+"    Let R2=128"
A$=A$+"    Anim 0, (1,2) (2,2) (3,2) "

Rem *** Check status of joystick...
A$=A$+"A:  If Joy1 & 1 Jump B"
A$=A$+"    If Joy1 & 2 Jump C"
A$=A$+"    If Joy1 & 4 Jump D"
A$=A$+"    If Joy1 & 8 Jump E"
A$=A$+"    Jump J"

Rem *** Update and check 'Y' co-ordinate...
A$=A$+"B:  Let R2 = R2 - 4"

```



```
A$=A$+"    If R2 < 0 Jump F"
A$=A$+"    Jump J"
A$=A$+"C:  Let R2 = R2 + 4"
A$=A$+"    If R2 > 235 Jump G"
A$=A$+"    Jump J"

Rem *** Update and check 'X' co-ordinate...
A$=A$+"D:  Let R1 = R1 - 4"
A$=A$+"    If R1 < 0 Jump H"
A$=A$+"    Jump J"
A$=A$+"E:  Let R1 = R1 + 4"
A$=A$+"    If R1 > 272 Jump I"
A$=A$+"    Jump J"

Rem *** Reset 'Y' co-ordinate...
A$=A$+"F:  Let R2 = 0"
A$=A$+"    Jump J"
A$=A$+"G:  Let R2 = 235"
A$=A$+"    Jump J"

Rem *** Reset 'X' co-ordinate...
A$=A$+"H:  Let R1 = 0"
A$=A$+"    Jump J"
A$=A$+"I:  Let R1 = 272"
A$=A$+"    Jump J"

Rem *** Pass co-ordinates to bob...
A$=A$+"J:  Let X = R1 + RA"
A$=A$+"    Let Y = R2"
A$=A$+"    Pause"
A$=A$+"    Jump A"

Rem *** AMAL Hardware Scroll program...

Rem *** Set initial value of screen...
B$=""      Let RA = 0"
```

```
Rem *** Add 2 to screen offset...
B$=B$+"A:  Let RA = RA + 6"

Rem *** Check that screen offset has not exceeded 320...
B$=B$+"    If RA > 320 Jump B"
B$=B$+"    Jump C"

Rem *** Reset screen offset if it has...
B$=B$+"B:  Let RA = 0"
B$=B$+"    Jump C"

Rem *** Pass 'RA' to screen offset 'X' co-ordinate...
B$=B$+"C:  Let X = RA"
B$=B$+"    Pause"
B$=B$+"    Jump A"

Synchro Off

Bob 1,160,128,1

Channel 1 To Bob 1
Channel 2 To Screen Offset 0

Amal 1,A$
Amal 2,B$
Amal On

Do
  Bob Clear

  Rem *** Run AMAL program...
  Synchro

  Bob Draw
  Screen Swap 0
  Wait Vbl
Loop
```

## Updating a rainbow effect

AMOS can also be used to update a rainbow effect by writing a different values into the 'X', 'Y' and 'A' registers. These registers are usually used to hold the co-ordinates and image of screen objects, but they have a slightly different use when applied to Rainbows. The 'X' register holds the value of the first colour in the rainbow, the 'Y' register contains the line on the screen (as a hardware co-ordinate) where the rainbow effect starts and the 'A' register holds the height of the rainbow in scan lines. Here's a slightly modified version of the 'Animated Rainbow' demonstration program from chapter 6 that uses AMAL to cycle through the rainbow under interrupt.

```
Rem *** Animated Rainbow Effect
Rem *** Filename - AMALRainbow.AMOS
```



```
Screen Open 0,320,256,2,Lowres
Flash Off : Curs Off
```

```
Rem *** Define AMAL program...
A$=""      Let R1 = 0"
A$=A$+"A:  Let R1 = R1 + 2"
A$=A$+"    If R1 > 191 Jump B"
A$=A$+"    Jump C"
A$=A$+"B:  Let R1 = 0"
A$=A$+"    Jump C"
A$=A$+"C:  Let X = R1"
A$=A$+"    Pause"
A$=A$+"    Jump A"
```

```
Set Rainbow 0,0,192,"", "", ""
```

```
Rem *** Set up copper list
COUNT=0
For R=0 To 15
    RGB=Val(Hex$(R)+"00")
    Rain(0,COUNT)=RGB
```

```
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next R
For R=15 To 0 Step -1
    RGB=Val(Hex$(R)+"00")
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next R
For G=0 To 15
    RGB=Val("$0"+Right$(Hex$(G),1)+"0")
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next G
For G=15 To 0 Step -1
    RGB=Val("$0"+Right$(Hex$(G),1)+"0")
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next G
For B=0 To 15
    RGB=Val("$00"+Right$(Hex$(B),1))
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next B
For B=15 To 0 Step -1
    RGB=Val("$00"+Right$(Hex$(B),1))
    Rain(0,COUNT)=RGB
    Rain(0,COUNT+1)=RGB
    COUNT=COUNT+2
Next B

Rem *** Turn on Rainbow effect...
Rainbow 0,0,0,280
```



```
Rem *** Assign AMAL channel to Rainbow...  
Channel 1 To Rainbow 0
```

```
Amal 1,A$  
Amal On  
Direct
```

# Sound and Music

- Built-in sound effects
- Sound samplers
- Sample banks
- Handling samples
- The Amiga's sound filter
- Music modules
- VU meters
- D-Sam extension
- Sample Bank Maker

**F**lashy graphics and great gameplay is all-important to any game, but you should never forget a game's sound effects. Although many view sound effects purely as cosmetic additions to a game, any magazine games reviewer will tell you that good sound effects can add a great deal of atmosphere. In many ways, sound effects are just as important in games programming as they are in the movies – imagine how different a film like 'Jurassic Park' would have been if it weren't for John Williams' tense sound track and those wonderfully blood-curdling sound effects! If Spielberg had opted for some rather wimpy growling effects sampled from a household cat, the Tyrannosaurus Rex wouldn't have been even half as frightening.

Back in the days when the Commodore 64 and Spectrum still ruled the roost, game sound effects were rather limited, but that certainly can't be said of the Amiga. Although Commodore still haven't touched the Amiga's sound chip after more than eight years (who knows, Commodore may have got around to releasing the infamous 'DSP' upgrade by the time you read this!), the good old Paula sound chip still ranks up there with the best of 'em.

So what makes Paula so special? Well, much of the credit goes to its ability to play sound samples. Indeed, the sound sample forms the foundations of the Amiga's sound capabilities, so Paula is ideally suited to play samples of anything from a roaring T-Rex to the bleating of a lamb. And, unlike certain other computers I could name, playing sound samples puts virtually no strain on the Amiga whatsoever, so your games can run at full speed even with the most complex sound sample booming from your monitor speakers!

Of course the Amiga isn't just limited to playing a single sound sample at any one time – no siree, it can play up to four different sounds at any one time and this has been used extensively by demo and game programmers to produce music using sampled instruments. Indeed, the Amiga's musical capabilities are so powerful that many professional musicians even use the Amiga for this very purpose – rumour has it that Paula Abdul, Renegade SoundWave and a whole bunch of well known indie bands use the Amiga in preference to dedicated sampling hardware.

## Sound sampling



**Built-in fx**

If you need a quick and easy method of incorporating some very simple sound effects into your games software, then there's no need to get bogged down with AMOS's sample handling capabilities. Francois Lionet (the author of AMOS) very cleverly built a number of sound effects directly into AMOS which provide the games programmer with a number of commonly used effects 'on the fly'. These are accessed using the 'Boom' (an explosion sound), 'Shoot' (a gun shot sound) and 'Bell' (a very simple bell sound). Try typing these three commands from AMOS's 'Direct' mode.

Obviously, these three sounds are far from earth-shatteringly exciting, so you'll be pleased to learn that you can make use of your own sound samples from directly within AMOS. Before you can go any further, however, you need to either find some pre-recorded sound effects (these are freely available in the Amiga public domain) or – better still – grab your own samples using a sound sampler. For the uninitiated amongst you, a sound sampler is an inexpensive piece of hardware that, when connected to the Amiga's parallel port, lets you literally 'record' sound directly from any sound source (a CD player is best). And, because the sound is held within your Amiga's memory as a series of numbers, once you've recorded some sound into your Amiga, you can edit it to your heart's content.

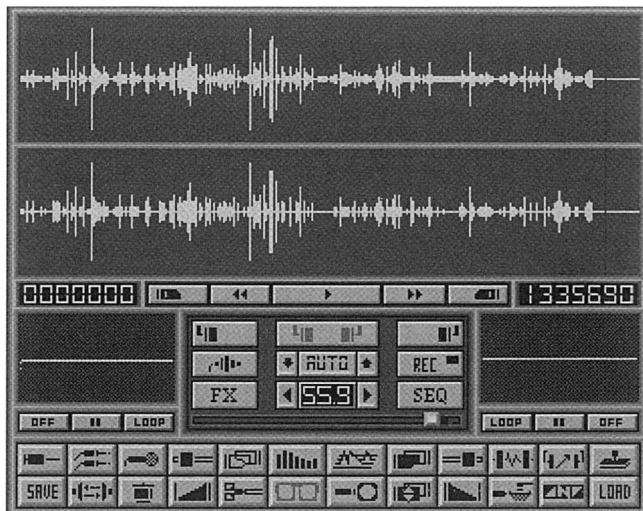
A word of warning, though. If you are intending to make money from your AMOS creation, then you must not use samples grabbed from copyrighted material – music and films, for example. If you do, then I can virtually guarantee you that you'll receive a visit from the boys in blue. If you really are desperate for good source material, then one of the many 'Sample CDs' that are available from companies such as 'Time + Space' (0442 870681) and 'AMG' (0252 717333) are a far safer bet. Time + Space, for example, produce a range of copyright-free samples from hit movies including 'Star Trek' and 'Terminator 2'.



**Copyrighted  
material**

Most hardware add-ons are terribly expensive, but sound samplers are certainly the exception. Indeed, they're so cheap these days that you can pick up a fairly decent unit for little more than £30. Of all the sound

*Buying a sound sampler will allow you to grab sound samples direct from any audio source. Once grabbed, they can be used directly within AMOS.*



samplers I've used during my many years as a magazine reviewer, however, my personal faves have to be New Dimensions' 'TechnoSound Turbo' (around £45) and MicroDeal's MIDI-compatible 'AMAS 2' (around £99). There are cheaper samplers available, but most of them leave a lot to be desired in the sound quality department. Buying a sound sampler is a bit like buying a can of baked beans – you may save yourself a few pence, but it's always worth spending a bit more for quality (you'll never catch me eating any brand other than Heinz!).



**TechnoSound  
Turbo,  
AMAS 2**



**Sample bank**

Once you've collected together the samples you need for your AMOS creation, you need to start by pulling them together into a 'sample bank'. It is possible to load sound samples directly into AMOS, but it's a bit technical to say the least. For this reason alone, it's far better to pull them together into a sample bank. A sample bank is very similar to a sprite bank, but for the obvious difference – whereas a sprite bank contains sprite images, a sample bank contains sound samples. Europress very kindly provide a very handy AMOS accessory program called the 'Sample Bank Maker' that allows you to create your own sample banks. AMOS 1.35 owners won't have this tool, but it's bundled free of charge with both AMOS Professional and Easy AMOS. Once again, the Sample Bank Maker is another damned good reason why you should consider upgrading. There are a number of PD alternatives, but none of them are as easy to use as Francois Lionet's original.

I won't bore you by talking too much about the Sample Bank Maker, however – if you're still not sure how to use it, then check out the end of this chapter for a full description of its many options.

Once you've created your sample bank, you need to start by loading it into your AMOS program using the 'Load' command. By default, the sample bank is automatically loaded into bank number 5, so it's best to keep this bank free if your program uses a lot of packed pictures.

Playing a sound sample is very easy indeed. All you need is the following command:

---

**Sam Play VOICE, SAMPLENUM, FREQUENCY**

**VOICE** The 'Voice' parameter is an optional parameter that specifically tells AMOS which of the Amiga's four sound voices the sample is to be played through. The parameter must be passed in the form of a four byte binary bitmap, with each byte representing a single voice. A bitmap of %0001, for example, would tell AMOS to play the sample on voice one. A bitmap of %1111, however, would tell AMOS to play the sample across all four voices.

**SAMPLENUM** The 'SampleNum' parameter is simply a number that tells AMOS which sample stored in the sample bank to play.

**FREQUENCY** The 'Frequency' parameter is an optional parameter that sets the replay speed of the sample. If you do not pass this parameter, the sample will be played at its default frequency (this is held within the IFF structure of the sample). However, passing a value lower than the sample's default frequency will cause the sample to be played at a lower pitch. A higher value will cause the sample to be played at a higher pitch.

Both the 'Voice' and 'Frequency' parameters are optional, so there's no reason whatsoever why you couldn't play a sample simply by passing the sample's sample bank number.

## The Amiga's built-in sound filter

Before we take a look at a demonstration listing, now is probably a good time to discuss the Amiga's sound filter. Built into every Amiga is a 'high pass' filter that filters out much of the high frequency 'noise' normally associated with 8-bit sound samples. Although this was permanently enabled on the early Amiga A1000's, every Amiga that Commodore has released since then offers the ability to turn this sound filter off, effectively increasing the frequency response of your sound samples. Once the filter has been turned off, you'll notice a far greater amount of high frequency sound in your samples, giving a much better sound quality. AMOS supports this feature too with a very simple command. Simply by adding the command 'Led Off' or 'Led On', the sound filter is turned off and on respectively.



### Dimming power light?

You'll notice that when the sound filter is turned off, the Amiga's power light will appear to dim – don't worry about this, since turning the sound filter off will do no damage to your Amiga whatsoever. Anyway, here's that demonstration listing that I promised you (complete with the 'Led Off' command!):

```
Rem *** Sample Replay Demonstration
Rem *** Filename - SamplePlay.AMOS

Rem *** Load sample bank..
Load "AMOSBOOK:AbkFiles/Instruments.ABK"

Rem *** Turn off sound filter...
Led Off

Locate 0,5
Centre "Press any key!"

Do
  A$=Inkey$
  SC=Scancode

  If A$<>""
```



```
If SC>0 and SC<14
    Rem *** Play sample 1 on voice 1...
    Sam Play %1,1
End If

If SC>15 and SC<28
    Rem *** Play sample 2 on voice 2...
    Sam Play %10,2
End If

If SC>31 and SC<43
    Rem *** Play sample 3 on voice 3...
    Sam Play %100,3
End If

If SC>48 and SC<59
    Rem *** Play sample 4 on voice 4...
    Sam Play %1000,4
End If
End If
Loop
```

---

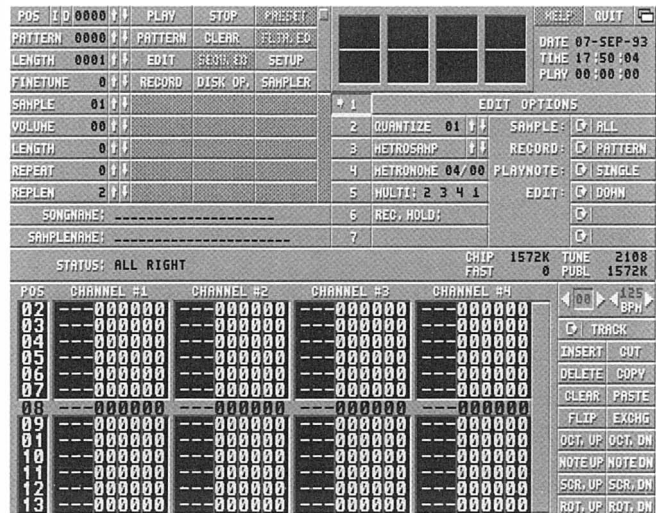
## Music modules

Great gameplay is the all-important factor in any game, but presentation isn't far behind. Even if your game is the most playable thing since 'Tetris', you still need that certain something to grab people's attention. Sure, great graphics certainly help, but you should never forget a game's soundtrack. Just like a game's sound effects, the music score that plays when a game's title screen is displayed can add a great deal to its atmosphere.

Steven Spielberg's 'Jurassic Park' is perhaps a good example here too – if you ever get a chance to watch the film again (who knows, it might be out on video by the time you read this), pay particular attention to the soundtrack playing in the background. Note how it considerably enhances the tension of the film – if you don't believe me, try turning the volume right down and see the difference it makes!



AMOS directly supports Sound Tracker-format 'Modules', allowing you to play music scores from within your AMOS creations with ease. What's more, most Sound Tracker clones are available for the price of a disk from Public Domain libraries, so you don't have to break the bank obtaining one.



Obviously most games don't play music when the game is being played, but even the simplest ditty can add a great deal of extra sparkle to your games. AMOS supports a number of sound track formats including its own 'Music bank' format and even Teijo Kinnunen's 'MED' format, but by far the most popular is the good old Sound Tracker 'module' format. AMOS' own format is perhaps the more versatile, but – due to reasons known only to the guys at Europress Software – the utility needed to convert Sound Tracker and MED format modules to AMOS bank format was never included with AMOS Professional. This isn't really surprising, however – with Sound Tracker clones becoming more and more diverse in the features that they offer (some, for example, now support 'Synthetic' instruments and even eight channels of sound), Sound Tracker module formats are changing too.

## Finding Sound Tracker modules

So how do you get your hands on an original Sound Tracker module? After all, there's very little point in using a Sound Tracker module that has already been used in a hundred other AMOS games! Well, the first option is to approach a friend who is musically inclined. If all your friends are totally tone deaf, however, you could try writing your own. What you'll need is a Sound Tracker clone such as 'ProTracker', 'NoiseTracker' or 'Audio Sculpture' (my personal favourite). The good

news is that – with the exception of ‘Audio Sculpture’ at least – virtually all Sound Tracker programs are public domain. That is, they’re available totally free, gratis and practically for nothing. If in doubt, contact a friendly Public Domain library (17Bit Software, for example) and they’ll almost certainly be able to supply you with a suitable program.

Once you’ve got your Sound Tracker program and you’ve written a module, you need to pull it into AMOS. Thankfully this is very easily indeed. All you need is the following command:

---

**Track Load** "FILENAME", BANKNUM



**FILENAME** The ‘Filename’ parameter is simply the filename (complete with full path information) of the module you wish to load.

**BANKNUM** Although a Sound Tracker module is not strictly an AMOS bank, AMOS still stores them into a memory bank. The ‘BankNum’ parameter should therefore be the number of a valid AMOS Bank.

With your Sound Tracker module loaded into an AMOS memory bank, all that remains is to play it. AMOS is very clever here – considering the complexities involved in playing music, you’ll be pleased to learn that AMOS does virtually all of the work away for you. For starters, all music scores are played under interrupt, so once you’ve set your module playing, you never have to worry about it again. The command to start a module playing is as follows:

---

**Track Play** BANKNUM



**BANKNUM** The ‘BankNum’ parameter tells AMOS which memory bank your module has been loaded into. If, for example, you had loaded your module into memory bank ‘6’, then you’d pass a value of 6 to start the module playing. As you can see, there’s virtually no reason whatsoever why you couldn’t have several modules stored in memory at once. You could, for example, have one module for the title screen, another for the high score screen and a rather solemn ditty for the ‘Game Over’ screen.

By default, AMOS will stop playing a module once it reaches the end of the tune, but you can force it to automatically play the module over and over again simply by adding the line ‘Track Loop On’. If you want to stop a module playing at any time, all you need is the ‘Track Stop’ command. Simple, eh! Here’s a very simple demonstration program that plays a Sound Tracker module written by my old mate, Dave Collins. Dave would like me to add that if any software houses out there are after a good freelance musician, then he’s available. Dave is also available for Church Fetes, supermarket openings and Barmitzvas (only joking!).

```
Rem *** Tracker Module Play Demonstration
Rem *** Filename - TrackPlay.AMOS
```



```
Rem *** Load Tracker module...
Rem *** Module Copyright © Dave Collins
```

```
Track Load "AMOSBOOK:Modules/Realthing.MOD", 6
```

```
Rem *** Turn off audio filter...
Led Off
```

```
Rem *** Play module...
Track Play 6
Track Loop On
```

```
Direct
```

---

## VU meters

If you’re one of these flash types that likes to own the latest in hi-fi equipment, then no doubt your stereo boasts what the techies call ‘VU meters’ (short for ‘volume meters’). As their name suggests, VU meters dynamically display the volumes of certain frequencies as music is being played. Although not technically true VU meters, Amiga demo coders have been quick to jump on the hi-fi bandwagon and many demos feature these all-important additions. AMOS programmers too can add VU meters to their AMOS creations thanks to the ‘Vumeter()’ function. The format of the function is as follows:

**VOLUME = Vumeter(VOICE)**



**VOLUME** The ‘Vumeter()’ function returns a value between 0 and 63 which defines the volume of the specified sound channel at the time that function was called. If the sound channel was completely silent, then a value of 0 will be returned. However, if the sound channel is booming its heart out at the highest possible volume, then a value of 63 will be returned.

**VOICE** The ‘Voice’ parameter tells AMOS which of the Amiga’s four sound channels the ‘Vumeter()’ function is to check. Until Commodore get around to releasing an upgraded sound chip, I’m afraid only four channels of sound can be played simultaneously. As a result, you should pass a value between ‘0’ and ‘3’.

The values returned by the ‘Vumeter()’ function can be used to create a myriad of different effects ranging from ‘Sound To Light’ converters to dancing sprites, but by far the most popular has to be the ‘VU Bar’ effect that displays four vertical bars that rise and fall in time to the music. Coding a VU bar procedure is quite complex, but you need not worry – below you’ll find a demonstration program that includes a procedure (called simply ‘VU’) that can be ripped out and used within your own programs. It’s a fairly adaptable little beauty – not only can you dictate the position where the VU bars are displayed, but you’re even given control over the width, height and spacing of the bars! If you feel like being flash, why not create a ‘Rainbow’ effect that changes the colours of the bars for different intensities?

```
Rem *** VU Meter Demonstration
Rem *** Filename - VUMeter.AMOS
```



```
Screen Open 0,320,256,2,Lowres
Flash Off : Curs Off : Cls 0
Palette $0,$F00
```

```
Track Load "AMOSBOOK:Modules/Realthing.MOD",6
```

```
Led Off
Track Play 6
Track Loop On

Dim VU(4)
Global VU()

Do
    _VU[80,50,40,200,2]

    Wait Vbl
Loop

Procedure _VU[X,Y,W,H,G]
    Rem *** _VU Procedure
    Rem *** X = X Position of VU Meter display
    Rem *** Y = Y Position of VU Meter display
    Rem *** W = Width of VU Meters
    Rem *** H = Height of VU Meters
    Rem *** G = Gap between each meter

    Rem *** Erase old VU Meters...
    Ink 0
    Bar X,Y To X+((W+G+1)*4),Y+(H+1)

    For A=0 To 3
        Rem *** Read value of VU meter...
        V=Vumeter(A)

        Rem *** Is channel silent?
        If V=0
            Rem *** If so, decrease old value...
            VU(A)=VU(A)-2
            If VU(A)<0
                VU(A)=0
            End If
        End If
    End For
End Procedure
```

```

Else
    Rem *** Otherwise, write new value to VU() array...
    VU(A)=V
End If

Rem *** Calculate offsets for drawing command...
OFFSET#=(H-((H+0.0)/64)*VU(A))+Y
XOFFSET=X+(A*W)+((G+1)*A)

Ink 1
Rem *** Draw VU Bar...
Bar XOFFSET,OFFSET# To XOFFSET+W,Y+(H+1)
Next A
End Proc

```

---

## D-Sam extension

If you're serious about your sampling, then you may want to invest in 'D-Sam', a powerful extension for AMOS that adds a multitude of handy sample-related commands to the AMOS instruction set. Written by 'AZ Software', D-Sam adds over 46 new commands that allow you to load multiple samples into memory (without having to worry about sample banks!) or even load and play samples direct from disk, effectively allowing samples limited only by the storage device that contains them to be used. You could, for example, sample an entire CD direct to disk and then play it back using D-Sam. D-Sam also supports samples that contain AudioMaster 'Loop tables'. Loop tables allow you to create complex musical scores by looping certain sections of a sample in sequence.

D-Sam offers a number of distinct advantages over AMOS's own sample replaying facilities, too. Unlike AMOS, which can only handle samples at rates of up to 29 KHz, D-Sam can play back samples at a maximum of 56 KHz which – believe it or not – is a higher sampling rate than even a CD player can handle. Obviously, though, you won't get the same sound quality as you'd get from a CD player (the Amiga's sound capabilities are only 8-bit, whereas CD players process samples at a 16-bit resolution). However, even if you play a sample at a lower rate, D-Sam offers 'software oversampling', a feature very similar to the

oversampling features available on most modern CD players. Oversampling can dramatically improve the sound quality of a sample by ‘smoothing it out’.

---

## Sample Bank Maker

As we saw earlier within this chapter, you need to create a sample bank in order to play sound samples within your AMOS programs. To make this task somewhat easier, Francois Lionet (the programmer of AMOS) very kindly included a handy accessory program called the ‘Sample Bank Maker’ with all versions of AMOS since Easy AMOS. Owners of AMOS 1.35 won’t actually have this tool on their AMOS program disks, but you can get a very similar tool by getting your hands on an AMOS updater disk. Better still, why not upgrade to AMOS Professional!

You can load the AMOS Sample Bank Maker from within AMOS Professional simply by selecting ‘Edit Samples’ from the ‘User’ pull down menu. Easy AMOS owners will have to load it into AMOS by clicking on ‘Load’ from the Easy AMOS menu strip and then selecting the file ‘Sample\_Bank\_Maker.AMOS’ when the Easy AMOS file requester appears.

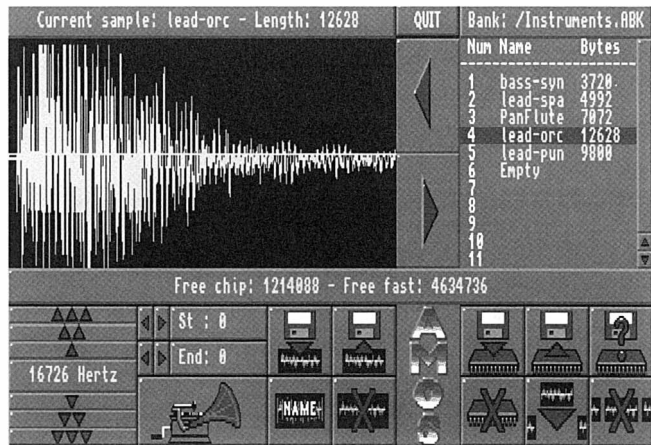
The Sample Bank Maker screen is split into three sections – the sample display window, the sample bank list and (running along the bottom of the screen), the Sample Bank Maker gadgets. The sample display window displays the current sample as a waveform display. You can fine-tune the start and end points of a sample simply by moving the mouse pointer either to the far left or to the far right of this display and then – whilst holding down the left mouse button – dragging the mouse pointer either left or right. As soon as you let go of the mouse button, the shortened sample will be played.

The Sample bank list displays the contents of the sample bank from top to bottom. Each sample is listed with its identification number (its position in the sample bank), name and its length in bytes. Samples can be transferred back and forth between the sample bank and the waveform display by clicking on the two arrow gadgets that separate these two windows.

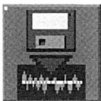


### Sample Bank Maker

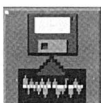
If you intend to use sampled sounds from within your AMOS programs, then you'll need to start by building up a sample bank using the Sample Bank Maker accessory program.



Below these two displays are the Sample Bank Maker gadgets which form the heart of the Sample Bank Maker. These gadgets allow you to load a sample and insert it into a sample bank, fine-tune the frequency of a sample, delete a sample and a whole lot more besides. Let's take a look at each gadget in turn:



**Load Sample** Not surprisingly, this option allows you to load a sample into the sample bank. Before you click on this gadget, however, you should click on the position in the sample bank where the sample is to be placed. AMOS automatically creates an 'Empty' slot in the sample bank for all new samples. The Sample Bank Maker supports two different types of sample – IFF samples and 'Raw' samples. If you load an IFF sample, then the Sample Bank Maker will also store the sample's replay frequency. Raw samples, on the other hand, do not have such information attached to them, so AMOS automatically sets the sample's replay frequency to 8 KHz (8363 Hz).



**Save Current Sample** If you need to extract a particular sample from a sample bank, then this is the option you need. By simply selecting the sample you wish to save from within the sample bank list and then clicking on this option, AMOS will display a file requester. Enter the filename, click on the 'OK' gadget' and the sample will be saved in IFF format.





**Delete Current Sample** Clicking on this option will instruct the Sample Bank Maker to remove the current sample from memory. Note that this does not remove the sample from the sample bank, however – it simply clears the waveform display so that no sound is displayed.



**Rename Sample** By default, the Sample Bank Maker will place each and every sample into the sample bank using the sample's original filename as an identifier. If you want to change the name of a sample, however, simply double-click on the sample you wish to change, click on this option and a requester will appear allowing you to edit the sample's name. Note that sample names are restricted to eight characters only.



**Tune Sample** These rather confusing gadgets allow you to adjust the frequency at which the current sample is played. The current frequency of the sample is displayed in the centre of the gadget and by clicking on the set of three gadgets above and below this frequency display, the frequency of the sample can be increased or decreased by varying amounts. The single arrow increases or decreases the frequency by 1 Hz, the double arrow by 10 Hz and the triple arrow by 100 Hz.



**Adjust Sample Points** If you don't own a dedicated sample editor (AudioMaster, for example), then this option comes in very handy if you need to isolate a section of a sample. Say, for example, you had a sample of someone saying 'Hello there' and you only wanted the 'Hello' bit – all you'd have to do is to adjust the 'End' point of the sample so that the 'there' bit is ignored.



**Play Sample** No matter how obvious your sample filenames may be, there's nothing better than actually being able to hear a sample. And, surprise, surprise, that's exactly what this option does.



**Insert Empty Sample** You may find that whilst building up your sample bank you need to insert a sample between two existing samples. Don't worry – you don't have to start from scratch! Just click on this option and an 'empty' sample will be inserted between the current sample and the sample immediately above the current sample.



**Delete Sample** The ‘Delete Sample’ option allows you to permanently remove a sample from the sample bank. Simply click on the sample you wish to delete, click on this option and a warning will pop up onto the screen asking you if you wish to remove the sample you have selected. Click ‘Yes’ to remove it and ‘No’ to cancel.



**Erase Sample Bank** If for some reason you’re not entirely happy with your sample bank, then you can zap it from memory simply by clicking on this option. Be very careful, however – unless you’ve previously saved it out to disk, it will be gone forever!



**Load Bank** The ‘Load Bank’ option allows you to load an existing sample bank into memory for further editing.



**Save Bank** If the sample bank has previously been either saved off to disk or loaded from disk, then clicking on this option will save the entire sample bank back to disk under its original filename. Note that the new file will completely overwrite the old sample bank.



**Save Bank As** If you want to save a sample bank to disk under either a different name or it hasn’t already been saved at least once, then this is the option you need. The ‘Save As’ option will bring up a file requester allowing you to specifically tell AMOS where and under what filename the sample bank should be saved.



# Games Programming

- Games programming principles
- The 'main game loop'
- Game types
- Optimising game code

**P**rogramming languages like AMOS can be applied to just about any type of programming project, ranging from databases and spreadsheets to paint programs and even educational software. Indeed, Europress themselves have used AMOS on numerous occasions to produce a whole range of ‘serious’ and educational packages including ‘MiniOffice’ and their ‘Fun School’ range of programs. Look in the PD libraries and you’ll find even more obscure AMOS creations such as Biorythmn calculators, pools checkers and more.

All well and good, but ask any AMOS coder what type of program they would really like to write and chances are you’ll get the same reply over and over again – games. The fact is, inside every applications programmer is a budding games programmer just bursting to escape! So why isn’t everyone writing games and earning the sort of mega-bucks that big names such as Dave Jones, Andrew Braybrook and Mev Dinc reap in? Much of the blame can perhaps be attributed to the fact that few amateur games programmers actually understand how a game works. That is, the internal workings of the game that actually make it tick.

Writing a program such as a utility is quite easy because they have a very linear form – a program to convert a picture from one image format to another, for example, has a fairly simple structure. All that needs to be done is to read in the image in its native image format, restructure the image data and then save it out in the new image format. Simple.

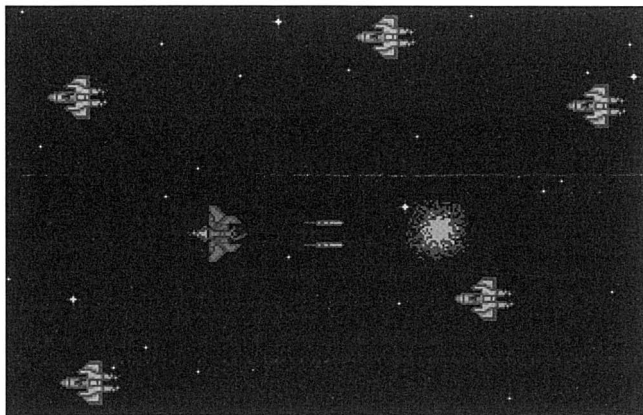
Games, on the other hand, are a different breed altogether. Because so many different things have to happen in just a single frame, it’s all too easy for most amateur programmers to get bogged down in code. Games tend to be very time-critical too – if a utility takes a couple of minutes to do its stuff, very few people are that concerned. As Lenin once said (or so I’m told), the end justifies the means. If you write a game that crawls along at a snail’s pace, however, it will be completely unplayable.



**Program  
speed**

To be perfectly honest, the only language that is really suitable for writing arcade games of true commercial quality is assembler (sacrilege!), the chosen language of professional games programmers, but AMOS can still do a pretty good job. Obviously if you try to code something like Team 17’s brilliant ‘Project X’ in AMOS, then expect it

*The secrets of games programming elude most programmers, but with the right knowledge you too could be churning out games with the best of 'em!*



to run at a snail's pace – but there's no reason whatsoever why you couldn't write a cut down version of Project X minus all the fancy thrills. As any games reviewer will tell you, at the end of the day the most important aspect of any game is its playability – Tetris, for example, is damned playable (I've certainly wasted quite a few hours bashing away with it) but even a slow programming language like Commodore's original AmigaBASIC could probably handle it.

We all dream of writing the sort of mega-games that have made programmers like Jez San household names, but you're in for a shock if you think that your initial efforts are going to be even close to the quality of commercial games such as StarGlider 2, Lemmings or Project X. Point number one – even as a beginner, you can produce some pretty good stuff with a little knowledge under your belt. But try to write a game that is beyond your programming talents and you'll soon get totally bogged down and very frustrated. Most amateur games programmers make this all too common mistake – after trying to undertake a game that is beyond their capabilities, most give up and throw in the towel completely. The moral of this point is simple – don't try to run before you can walk. Always start with a simple idea and then once you've honed your skills,, move onto something a bit more elaborate.



**Be  
conservative!**

Another very important aspect of the game programmer's art is code optimisation – that is, trying to make your code run as fast as possible. Most modern arcade games are updated at a rate of 50 times per second

so your entire game code has to be executed within this time if your game is to remain smooth. By constantly improving the performance of a given routine, you can squeeze more and more spare time from the Amiga's processor that could be used for extra effects such as parallax scrolling, animated copper bars etc.

Games programming has always seemed something of a mystical art grasped only by the chosen few, but in fact games aren't as complex as you might think. Fact is, all games are based around pretty much the same theory. Once you have rasped a few relatively simple principles and you feel confident enough to put it into practice, you'll be churning out games faster than you could possibly imagine! Simple shoot 'em ups are particularly easy – once you understand how a game is structured, it's quite possible to write quite a playable shoot 'em up along the lines of Project X in an afternoon!

---

## The main game loop

Looking at a game in its most simplistic form, all games are based around a loop that commercial game programmers call the 'main game loop'. The sole purpose of this loop is to perform all the functions that are required to make the game run – in an arcade game like Project X, for example, the main game loop would read the player's joystick and move the player's ship in response, update the positions of all aliens and missiles, check for collisions and update the score accordingly. Each time the loop is completed, the screen is redrawn once, so the faster you can get AMOS to execute this loop, the faster your game will run.

There is a point where the screen refresh rate of both your monitor and Amiga won't be able to keep up with your game, so there's little point redrawing the screen any faster than the Amiga can refresh the screen. On a PAL Amiga, for example, the screen is refreshed a maximum of 50 times per second, so there's little point redrawing the screen any faster than this. Although AMOS is pretty rapid, you're unlikely to find this a problem unless you code large sections of your game in assembler. Even then, you should tie screen redraws in with the vertical blanking period of the screen using the AMOS 'Wait Vbl' command. If you don't use 'Wait Vbl', any objects that you have moving on screen will appear to

flicker terribly and the entire game will be out of sync with the Amiga's screen refresh hardware.

So what does the main game loop actually do? Well, the best way to understand this bit of programming trickery is to imagine a game slowed down so that the main game loop is performed just once per second (most well-programmed commercial games perform their main game loop every video frame – that's fifty times per second!). Every time the main game loop is performed, every aspect of the game is updated just once – the player's ship along with any aliens and missiles are moved and any collisions are acted upon. If you were to run the main game loop just once, very little would appear to happen – you might see a couple of sprites move a couple pixels and then stop, but little else. It's not until the main game loop is run continuously that the game springs to life.



### Main game loop

Let's consider a relatively simple game like a shoot 'em up by taking a look at its main game loop in psuedo code form:

#### Start of Main Game Loop

```
Move alien spaceships
Move player's missiles
Move alien's missiles
```



```
Have any of the alien's fired a new missile?
  Generate new missile sprite
  Play the sound of a missile being fired
```

```
Has the joystick being pushed?
  Has it being pushed left?
    Move player's ship left by 'n' pixels
  Has it being pushed right?
    Move player's ship right by 'n' pixels
  Has it being pushed up?
    Move player's ship up by 'n' pixels
  Has it being pushed down?
    Move player's ship down by 'n' pixels
```



```
Has the joystick fire button being pressed?  
  Generate new missile sprite  
  Play the sound of a missile being fired
```

```
Have any of the player's missiles collided with aliens?  
  Remove alien sprite  
  Remove missile sprite  
  Generate explosion sprite  
  Play the sound of an explosion  
  Add 'n' points to player's score
```

```
Have any alien missiles collided with the player's ship?  
  Remove player's sprite  
  Remove alien missile sprite  
  Generate explosion sprite  
  End Game!!!
```

```
Redraw all sprites onto screen  
Swap physical and logical screens  
Wait for vertical blank before proceeding  
Jump back to start
```

As you can see, the main game loop of a game consists of quite a few decisions that control the flow of the program code. Any code that is not embedded inside a decision will be performed every time the main game loop is performed. Code that is embedded inside a decision will only be performed, however, if the result of the decision is true. Take the firing of a missile, for example. Obviously you only want a missile to be fired when the player presses the joystick fire button, so the first thing that is done is to check whether the fire button is being pressed (using the 'Joy(1)' function). If, after checking the fire button, you find that it hasn't been pressed, then there's little point in running the code that handles this particular aspect of the game.

As you can probably appreciate, there are a lot of steps within the main game loop that may not always be performed. If the results of any of the decisions made within the loop are found to be false (the player's ship hasn't collided with a missile, for example), then huge chunks of the

game's code will not be performed. Splitting up your code in this way (programmers call this 'modular' code) not only makes your game's code more readable, but it can also make your game run considerably faster.

We've covered some pretty heavy theory over the last few pages or so, so we'll put all that theory into practice over the remaining chapters by taking a look at how to write five of the most popular types of game – shoot 'em ups, platform games, maze games, 'Dungeon Master' clones and the good old graphic adventure.

---

## Optimising your games

AMOS turns in some pretty impressive code performance ratings, but it will never be anywhere near as fast as pure assembly language, the choice for commercial games programmers. There are ways, however, of squeezing that extra spurt of speed from your AMOS code.

- 1** Europress kindly bundle a runtime system with all versions of AMOS including Easy AMOS and the latest release, AMOS Professional. Although these runtime systems allow you to run your AMOS creations without having to load the AMOS interpreter, your programs will still run at the same speed. If you want to get your games running as fast as possible, then you should seriously consider purchasing the AMOS Pro Compiler. Because the compiler produces a machine code version of your program, the resulting code (particularly math-intensive routines) will run considerably faster.
- 2** AMAL is all fine and dandy for very simple programs, but it can create synchronisation problems with larger games – I've even heard horror stories of AMAL code crashing programs once they've been compiled! If you insist on using AMAL (most AMAL programs run no faster than their AMOS equivalents once compiled!), then it's well worth switching of the AMAL interrupt system (using the 'Synchro Off' command) and then running all your AMAL programs directly (with the 'Synchro' command).
- 3** AMOS's blitter object handling routines are very fast when compared to the likes of GFA Basic and AmigaBASIC, but they're still rather slow. If

If you want your games to run at peak performance, then you need to buy yourself a copy of the AMOS Compiler.



your game uses two or more bobs, then you should seriously consider switching off AMOS's automatic bob redrawing feature (using 'Bob Update Off') and then redrawing all bobs 'en masse' with the 'Bob Clear' and 'Bob Draw' commands. This will produce a considerable increase in code performance as all your bobs will be drawn onto the screen in a single blitter operation.

It's well worth handling the process of updating a double buffered display yourself too by switching AMOS's 'AutoBack' facility from its default setting ('3') to AutoBack mode '1'. When you need to swap the physical and logical screens, just add the line 'Screen Swap'.

- 4 It's very tempting to write games that make full use of the PAL display, but there's a very good reason why most games programmers still write games in NTSC resolutions – redrawing those extra 56 lines eat up valuable processor time. Try to keep your game screens as small as possible – even if you knock your game screen down from 256 vertical lines to 200 vertical lines, a considerable speed increase will be evident.
- 5 If you're compiling your game, always turn off the compiler's 'Runtime Error Checking' facility. Runtime error checking eats up valuable system cycles which can better be used by your game. Obviously this means that you'll have to playtest your game a lot more thoroughly to ensure that your game doesn't crash, but then this is a damned good habit anyway!

- 6 32-colour game screens look very nice indeed, but they're also very slow to update. You should therefore try to keep the depth of your screen as low as possible. Every extra bitplane that the Amiga's blitter has to work on will cause a slight decrease in code performance.

When designing blitter objects, you should always try to design them so that they use the first set of colours in a game's screen palette. Another good idea is to restrict the depth of the bobs in your sprite bank so that only the bitplanes that the bobs use are included in the sprite bank. Not only will this reduce the size of your sprite bank, but AMOS can plot them onto screen a lot faster.

- 7 If you're finding it hard to nail down the procedure that is slowing down your game, then a good tip is to insert the line 'Doke \$DFF180,\$RGB' (where '\$RGB' is a valid hex colour value) between each procedure in your main game loop, each with their own unique '\$RGB' value. When you run your program, the background colour will change at several different vertical positions, giving you a sort of psuedo-graph of the time required to run each procedure. If a section of colour is considerably larger than all the others, then you know exactly which procedure is causing the speed problems.
- 9 Keep it simple! AMOS may be fast, but it's still a Basic programming language at the end of the day. Although it can handle simple arcade games, don't expect it to run your AMOS version of 'Project X'. As any game reviewer will tell you, the most important aspect of any game is its gameplay – something that even the simplest games can have in abundance (check out 'Tetris' if you don't believe me).



# Shoot 'em ups

- Moving the player's ship
- Handling aliens
- Firing missiles
- Explosions

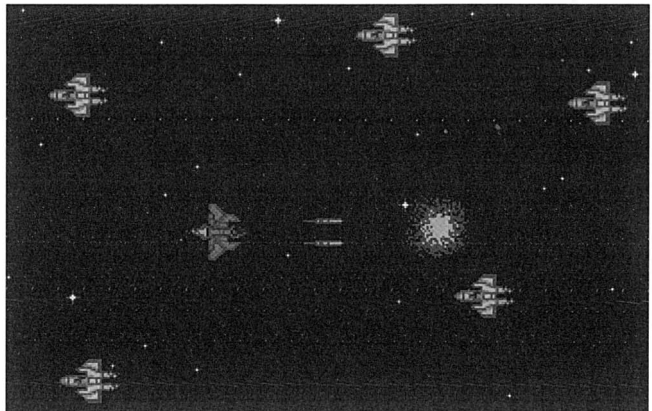
**W**e've already covered virtually all the theory behind shoot 'em ups both in the last chapter and previous chapters. Neary all the routines that you'll ever need to get a shoot 'em up running can be found somewhere in this book, so I won't say too much about the theory behind writing a decent shoot 'em up.

Even the pseudo code for a complete shoot 'em up can be found at the beginning of the last chapter, so there's little point in boring you with information that we've already covered. But before we dive into the listing for the simple demonstration game that you'll find on the disk, ask yourself the following question – what makes a good shoot 'em up? And, with so many games of this type available, what sort of ingredients does a shoot 'em up need to make it a cut above the rest?

Well, first and foremost, a shoot 'em up must be fast-paced – all good shoot 'em ups push the player's abilities to the limit by testing their reactions and their ability to stay cool in a tight spot. If your game runs so slowly that the player can simply move around the screen blasting aliens without batting an eyelid, it's hardly going to be challenging.

Secondly, a shoot 'em up must be violent. OK, so people like Mary Whitehouse and most of the tabloid press are dead set against violence in computer games, but it's the satisfaction of literally blasting your way through hordes of aliens that makes the shoot 'em up so appealing. Perhaps this explains why shoot 'em ups that allow the player to collect

*Programming arcade games is surprisingly simple providing you've got a little bit of basic knowledge under your belt.*



'power ups' are so popular – the more power ups you get, the more aliens you can kill in one shot. I know that this sort of thing certainly appeals to me – there's no nothing that can compare to that feeling of having such a deadly array of weapons on your spaceship that even the toughest of aliens are reduced to space dust with a single shot!

Lastly, a shoot 'em up must continue to be challenging. If you've written a shoot 'em up where every wave of aliens behave pretty much the same, the old boredom factor will be very high indeed. Don't make your shoot 'em up too difficult however – that's the second easiest way of losing the interest of a games player!

Anyway, that's enough of the theory – let's take a look at a listing for a playable shoot 'em up game. It's nothing special, so don't expect a game that blows 'Project X' out of the water. If you're feeling adventurous, why not have a go at adding a few extra features such as making the attacking hordes of aliens shoot back. And how about adding some 'power ups' and smart bombs. Good luck!

```
Rem *** Shoot Em Up Demo
Rem *** Filename - ShootEmUp.AMOS
Rem *** This game will not work under Easy AMOS!
```



```
Rem *** AMAL Hardware Scroll Routine...
SCRL$=""          Let RA = 0"
SCRL$=SCRL$+"A:  Let RA = RA + 1"
SCRL$=SCRL$+"    If RA > 320 Jump B"
SCRL$=SCRL$+"    Jump C"
SCRL$=SCRL$+"B:  Let RA = 0"
SCRL$=SCRL$+"C:  Let X = RA"
SCRL$=SCRL$+"    Pause"
SCRL$=SCRL$+"    Jump A"

Rem *** AMAL Joystick control routine...
JY$=""           Let RB = 40"
JY$=JY$+"       Anim 0, (1,2) (2,2)"
JY$=JY$+"       Let RC = 100"
```



```

JY$=JY$+"   A:  If J1 & 1 Jump B"
JY$=JY$+"       If J1 & 2 Jump D"
JY$=JY$+"       If J1 & 4 Jump F"
JY$=JY$+"       If J1 & 8 Jump H"
JY$=JY$+"       Jump W"
JY$=JY$+"   B:  Let RC = RC - 3"
JY$=JY$+"       If RC < 20 Jump C"
JY$=JY$+"       Jump W"
JY$=JY$+"   C:  Let RC = 20"
JY$=JY$+"       Jump W"
JY$=JY$+"   D:  Let RC = RC + 3"
JY$=JY$+"       If RC > 180 Jump E"
JY$=JY$+"       Jump W"
JY$=JY$+"   E:  Let RC = 180"
JY$=JY$+"       Jump W"
JY$=JY$+"   F:  Let RB = RB - 3"
JY$=JY$+"       If RB < 20 Jump G"
JY$=JY$+"       Jump W"
JY$=JY$+"   G:  Let RB = 20"
JY$=JY$+"       Jump W"
JY$=JY$+"   H:  Let RB = RB + 3"
JY$=JY$+"       If RB > 280 Jump I"
JY$=JY$+"       Jump W"
JY$=JY$+"   I:  Let RB = 280"
JY$=JY$+"       Jump W"
JY$=JY$+"   W:  Let X = RB + RA"
JY$=JY$+"       Let Y = RC"
JY$=JY$+"       Pause"
JY$=JY$+"       Jump A"

```

```
Dim MISSILE(2,3)
```

```
Rem *** Missile Data structure
```

```
Rem *** Missile(n,0) = Status (0=Off 1=Fired)
```

```
Rem *** Missile(n,1) = Missile X co-ordinate
```

```
Rem *** Missile(n,2) = Missile Y co-ordinate
```

```
NUMALIENS=5 : Rem *** Number of alien spaceships...
```

```

Dim ALIEN(NUMALIENS,4)
Rem *** Aliens Data structure
Rem *** Alien(n,0) = Alien X co-ordinate
Rem *** Alien(n,1) = Alien Y co-ordinate
Rem *** Alien(n,2) = Alien speed
Rem *** Alien(n,3) = Status (0=Fine 1=Exploding)

Global MISSILE(),MISSILEDELAY,ALIEN(),ALIENFRAME,FRAMEDELAY
Global SCRL$,JY$,SCROFFSET,SHIPX,SHIPY,DEAD,SCORE,NUMALIENS
Global BANG$

Rem *** Initialise game...
_GAMEINIT

```

Like all games, there's a fair amount of setting up to be carried out before the game enters the main loop. Our shoot 'em up uses AMAL to control both the screen scrolling and the movement of the player's ship. Two AMAL programs are required which are placed into the string variables 'SCRL\$' and 'JY\$'. In order to keep both programs running in sync, the current screen offset setting which is updated by the 'SCRL\$' program is stored into the global AMAL variable 'RA' which is used to position of the player's ship correctly in relation to the scroll. The ship control program ('JY\$') saves the horizontal ('X') position of the ship into a global AMAL program too – 'RB'. This is used later in the game to position the missiles that the player's ship can fire relative to the ship's position.

Two data structures are set up at the beginning of the game – ALIEN() and MISSILE() – which control the attacking alien spacecraft and the player's missiles accordingly.

```

Repeat
  Rem *** Main Game Loop...

```

```

  Bob Clear
  Synchro

```



```
Rem *** Get screen offset value from AMAL...
SCROFFSET=Amreg(0)

Rem *** Get position of ship from AMAL...
SHIPX=Amreg(1)
SHIPY=Amreg(2)

_CHECKCOLLISIONS
_CHECKFIRE
_MOVEALIENS
_MOVEMISSILES

Bob Draw
Screen Swap 0
Wait Vbl
Until DEAD=1
End
```

Here's the main game loop in all its AMOS glory – not very exciting, is it? Because both the running of AMAL programs and bob updates are handled by the programmer, we have to manually remove all bobs from the screen before proceeding. Once this is done, both our AMAL programs are run once. Once the screen has been scrolled and the position of the player's ship has been updated, several values are read from our AMAL programs and stored into variables. These will be used to correctly position both the alien bobs and the player's missiles.

Now the fun really starts. Four procedures are executed in order to make the game run – `_CHECKCOLLISIONS` (which checks for collisions between the player's sprite and the aliens and collisions between the player's missiles and the aliens), `_CHECKFIRE` (which generates the missile sprites when the player presses the fire button), `_MOVEALIENS` (which updates the positions of the attacking alien spacecraft) and `_MOVEMISSILES` (which moves the player's missiles). Finally, all the bobs are drawn onto the screen which is then swapped into view.



```
Procedure _GAMEINIT
  Screen Open 0,640,200,16,Lowres
  Flash Off : Curs Off : Cls 0
  Screen Display 0,128,48,320,200
  Hide

  Rem *** Load graphic and sound files...
  Load Iff "AMOSBOOK:Pictures/CosmoBlastBackground.IFF"
  Screen Copy 0,0,0,320,200 To 0,320,0

  Load "AMOSBOOK:AbkFiles/CosmoShips.ABK"
  Get Sprite Palette
  Make Mask 3

  Load "AMOSBOOK:AbkFiles/CosmoSoundFX.ABK"

  Double Buffer : Autoback 1
  Bob Update Off

  Rem *** Turn on player's ship...
  Bob 10,40,128,1

  Rem *** Initialise AMAL...
  Synchro Off
  Channel 0 To Screen Offset 0
  Channel 10 To Bob 10

  Amal 0,SCRL$
  Amal 10,JY$
  Amal On

  _INITALIENS
End Proc
```

The `_GAMEINIT` procedure is called at the start of the program to set up the game screen, load in the background graphics and the sprite and sample banks. The player's ship bob is created and our two AMAL programs are assigned to the screen and the bob we've just created. After

turning on the two AMAL channels that we assign to our AMAL channels, the procedure then jumps to the ‘\_INITALIENS’ procedure that initialises the first wave of attacking aliens.

```
Procedure _INITALIENS
```

```
  FRAMEDELAY=0
```

```
  ALIENFRAME=4
```

```
  For C=0 To NUMALIENS-1
```

```
    ALIEN(C,0)=320
```

```
    ALIEN(C,1)=Rnd(180)
```

```
    ALIEN(C,2)=Rnd(7)+2
```

```
    ALIEN(C,3)=0
```

```
    Bob C,ALIEN(C,0)+SCROFFSET,ALIEN(C,1),ALIENFRAME
```

```
  Next C
```

```
  Anim On
```

```
End Proc
```



The movement of the attacking alien spacecraft is hardly particularly challenging, but it does work very well. The \_INITALIENS procedure initialises the first wave of aliens by randomly calculating both their vertical positions and their speed. When run, the aliens will appear to fly past the player’s ship at different speeds. As each ship is initialised, the bob for that ship is turned on and placed at the screen position held within its initialised data structure.

```
Procedure _CHECKFIRE
```

```
  MISSILE=-1
```

```
  Inc MISSILEDELAY
```

```
  Rem *** has fire button been pressed?
```

```
  If Joy(1) and 16
```

```
    Rem *** Have guns had time to recharge?
```

```
    IF MISSILEDELAY>20
```

```
      Rem *** Are missiles still active?
```



```
For C=0 To 1
  If MISSILE(C,0)=0
    MISSILE=C
  End If
Next C
MISSILEDELAY=0

Rem *** Fire missile!
If MISSILE>-1
  MISSILEX=X Hard(SHIPX)
  MISSILEY=Y Hard(SHIPY)
  Sprite MISSILE*4,MISSILEX,MISSILEY,3

  MISSILE(MISSILE,0)=1

  Rem *** calculate position of missile
  Rem *** relative to spaceship position
  MISSILE(MISSILE,1)=SHIPX+10
  MISSILE(MISSILE,2)=SHIPY+7

  Sam Play 1
End If
End If
End If
End Proc
```

The `_CHECKFIRE` procedure handles the task of initialising new missiles when the player presses the fire button. The routine starts by checking whether exactly twenty loops have passed (this is indicated by checking the value of the variable 'MISSILEDELAY'). If enough time has passed, the routine then checks to see whether a missile is available for firing by entering a loop that checks the value of the first element in each missile's data structure ('Missile(n,0)'). If a missile is found, the missile's data structure is set up and a missile sprite is drawn onto the screen. Finally, a sampled sound of a missile is played using the 'Sam Play' command.

```

Procedure _MOVEMISSILES
  For C=0 To 1
    Rem *** Is missile active?
    If MISSILE(C,0)=1

      Rem *** Increase X position of missile...
      MISSILE(C,1)=MISSILE(C,1)+8

      Rem *** Has missile left screen?
      If MISSILE(C,1)>320
        Rem *** Turn off missile...
        Sprite Off C*4
        MISSILE(C,0)=0
      Else
        Rem *** Redraw missile...
        MISSILEX=X Hard(MISSILE(C,1))
        MISSILEY=Y Hard(MISSILE(C,2))
        Sprite C*4,MISSILEX,MISSILEY,3
      End If
    End If
  Next C
End Proc

```



The `_MOVEMISSILES` procedure is pretty self-explanatory – its sole role in life is to update the positions of any missiles that may have been fired by the player. It has a dual role, however – not only does it increase the ‘X’ screen position of each missile, but it also checks whether a missile has left the screen (indicated by a missile ‘X’ position greater than 320). If the missile has left the screen, the data structure associated with that missile is reset so that the missiles becomes available and the missile sprite is removed from the screen.

```

Procedure _MOVEALIENS
  Rem *** Update alien animation...
  If FRAMEDELAY>2
    FRAMEDELAY=0
    If ALIENFRAME=4

```



```
        ALIENFRAME=5
    Else
        ALIENFRAME=4
    End If
End If
Inc FRAMEDELAY

For C=0 To NUMALIENS-1
    If ALIEN(C,3)=0
        Rem *** Move alien spaceships...
        ALIEN(C,0)=ALIEN(C,0)-ALIEN(C,2)

        Rem *** Check that alien has not left screen...
        If ALIEN(C,0)<-40
            ALIEN(C,0)=320
            ALIEN(C,1)=Rnd(180)
            ALIEN(C,2)=Rnd(7)+4
        End If
        Bob C,ALIEN(C,0)+SCROFFSET,ALIEN(C,1),ALIENFRAME
    Else
        Inc ALIEN(C,3)
        Rem *** Has explosion run its course?
        If ALIEN(C,3)>10
            ALIEN(C,0)=320
            ALIEN(C,1)=Rnd(180)
            ALIEN(C,2)=Rnd(7)+4
            ALIEN(C,3)=0
        Else
            Rem *** Update explosion...
            Bob C,ALIEN(C,0)+SCROFFSET,ALIEN(C,1),6
        End If
    End If
Next C
End Proc
```

The `_MOVEALIENS` procedure is responsible for updating the screen positions of all the aliens. It starts by updating the animations of the aliens. Each alien only uses two frames, so the animation frame number



is shared between all the aliens. Each alien data structure has a value associated with it that dictates whether the spaceship has been destroyed (a value of 1) or whether the alien is still moving on the screen (a value of 0). If the ship is still active, its position is updated by subtracting its speed from its current 'X' screen position. If the alien leaves the screen (indicated by a 'X' value less than zero), a new alien is generated using exactly the same technique used in the '\_INITALIENS' procedure.

If the alien has been destroyed (indicated by a status value of 1), an image of an explosion is displayed in the place of the normal alien image. This explosion graphic is held on the screen using a counter that is held in the alien's 'speed' variable. If the counter reaches a value greater than 10, the explosion has run its course and a new alien is generated in its place.

#### Procedure \_CHECKCOLLISIONS

```

Rem *** Has missile 1 hit an alien?
STATUS=Spritebob Col(0,0 To NUMALIENS-1)
If STATUS=-1
  Rem *** Which alien did it hit?
  For C=0 To NUMALIENS-1
    If ALIEN(C,3)=0
      If Col(C)=-1
        ALIEN(C,3)=1
        MISSILE(0,0)=0
        Sprite Off 0
        Rem *** Add 1 to score
        Inc SCORE
      End If
    End If
  Next C
  Sam Play 2
End If

Rem *** Has missile 2 hit an alien?
STATUS=Spritebob Col(4,0 To NUMALIENS-1)
If STATUS=-1

```



```
For C=0 To NUMALIENS-1
  If ALIEN(C,3)=0
    If Col(C)=-1
      ALIEN(C,3)=1
      MISSILE(1,0)=0
      Sprite Off 4
      Inc SCORE
    End If
  End If
Next C
Sam Play 2
End If

STATUS=Bob Col(10,0 To NUMALIENS-1)
If STATUS=-1
  Sam Play 2

  Rem *** Enable this line to end game when player hit!
  DEAD=0
End If
End Proc
```

The final procedure is responsible for detecting collisions between the player's sprite and the aliens and collisions between the player's missiles and any aliens that are unlucky enough to stand in their way. Two separate checks are necessary to cover the two different missiles that could be on the screen at any time – if either of them strikes an alien, the alien's status is set to one (indicating that it is exploding), the missile is turned off by setting its status to 0, the sprite is turned off and finally, the player's score is increased.

The last collision check is carried out to see whether the player's ship has collided with an alien. Although I've left the code out, you could quite easily add your own code that either makes the player lose a life or – worse still – ends the game.



# Maze games

- Drawing the maze
- Detecting Walls
- Picking up objects
- Intelligent 'baddies'
- Dungeon Master clones

**M**aze games come in all shapes and sizes, ranging from dungeon romps like ‘Gauntlet’, to age-old arcade classics like ‘PacMan’. Although they may seem very different indeed, the gameplay and indeed the programming techniques involved are very similar. What’s more, even state of the art 3D exploration games like FTL’s ‘Dungeon Master’ and ‘Eye of the Beholder’ share much of their code with the good old maze genre of game.

Before we get too heavily bogged down with Dungeon Master-style games, however, let’s start at the very beginning with a look at the traditional maze game, where the maze is viewed from a bird’s eye view. If you studied the ‘Screen Blocks’ listing in chapter 8, then you’ll already have a pretty good idea of how best to draw up a maze on the screen. Using tiny rectangular graphic blocks drawn in DPaint and then placed into an ‘Icon Bank’ using the AMOS Object Editor, you can draw up a very complex maze that uses very little memory indeed. What’s more, because each screen takes up only 1 or 2K, there’s no reason whatsoever why you couldn’t add hundreds of different screens with ease.

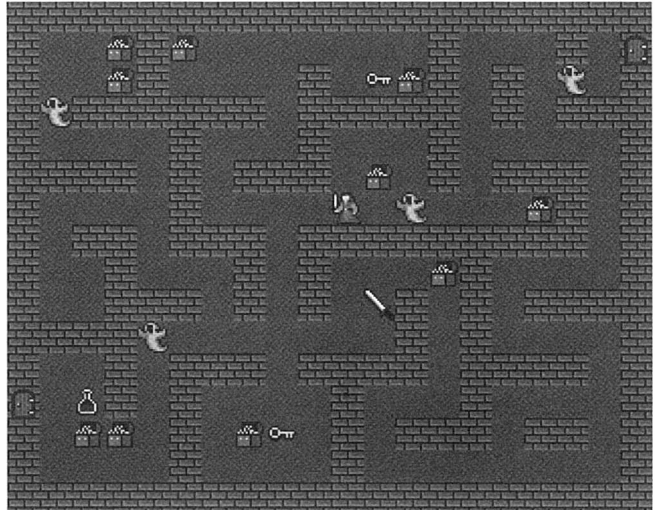
There is, however, still a lot of extra code that needs to be added to turn our ‘Screen Blocks’ listing from chapter 8 into a maze game. Think about the following – how do you move the player’s sprite around the maze without it walking through walls? What’s more, how do you add baddies to your game that wonder around the maze looking for the player’s sprite? This isn’t as simple as you might think – after all, if the game is to be challenging, then the baddies need to have a certain amount of intelligence!

Upon first inspection, the most obvious way to control the player’s movement around the maze is to use collision detection, but – believe it or not – there is a considerably easier way that is virtually foolproof. If you’ve studied the ‘Screen Blocks’ listing, then you’ll have noticed that the maze itself is held as nothing more than a series of numbers that are used to identify which screen blocks are to be pasted at certain screen positions. A value of ‘1’, for example, will paste a ‘ground’ graphic, a value of ‘2’ will paste a ‘wall’ block and a value of ‘3’ will paste a ‘door’ block etc. Although this technique will not produce random



**Easier collision  
detection**

*OK, so our maze game doesn't look that exciting, but believe it or not the code covered in this chapter could be used to produce anything from a Pacman game to a full-blown Gauntlet clone!*



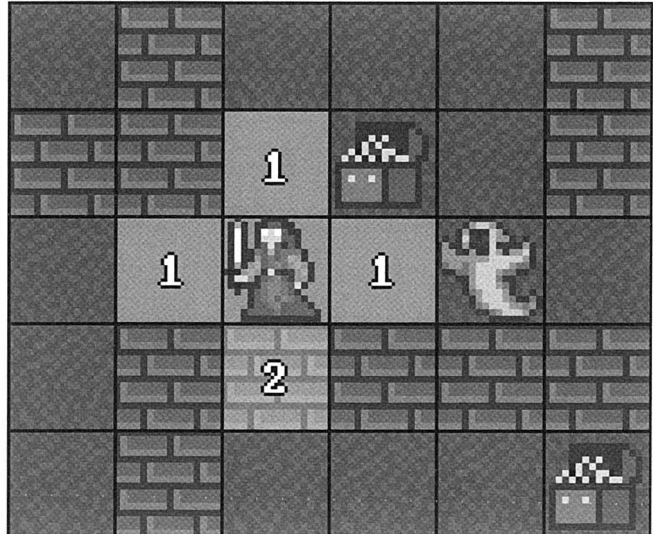
mazes, it does have one major advantage – because the mazes can be defined by the programmer, we can also use the same data as an alternative to using collision detection.

As the maze is essentially a 20 x 16 grid containing nothing more than numbers between 1 and 5, all we need to do is to treat the player's movement in the game as a series of steps through this grid expressed as a set of co-ordinates. For example, if the player was to be placed at position (2,2), he could theoretically move (in a single step) horizontally and vertically up, down, left or right to positions (2,1), (2,3), (1,2) or (3,2) respectively.

Getting this routine up and running is surprisingly simple. All you need to do is check the status of the joystick once during the main game loop – if the joystick is pushed up, then decrement the X co-ordinate of the player's position in the map, increment it if the joystick is pushed down and so on. Now this is all fine and dandy, providing that there are no walls in the way. If there are, the player's sprite will appear to walk straight through them as if it were a ghost. So how do we stop this from happening?

When the joystick is tested, the first thing that the movement routine does is to check whether the grid position that the character would move

The player's movement routine checks the data held in the map array to see if the square that the player is attempting to move to is a floor block.



to is actually an empty space. It does this by checking though a dimensional array that holds the same maze data that is used to draw the maze. Let's say, for example, that the player's sprite is at position (2,2) and the player has attempted to move the position to (2,1). If there is a wall there, the players sprite should not move. To achieve this the movement routine would calculate the theoretical new position and then check the maze data array to see if the value held within (2,1) is one (designating a floor block). If it isn't then the player's joystick input is simply ignored resulting in the sprite staying in its original position. If a value of 1 is found, however, then the player's position is updated accordingly.



You're probably thinking that this technique would result in some rather jerky movement – after all, the player's character is moving sixteen pixels in a single turn. Not so with some clever programming, however. All you need to do is to add a 'step' variable to the player sprite's data structure and then increment this until the entire movement step is completely. Until the 'step' variable reaches a maximum value, all other joystick inputs are ignored.

---

## Adding baddies

Now we have our hero happily running around the maze, all we need are some baddies to chase after him (or her). Movement of the baddies is not a problem – all you need to do is to adapt the routine you use for the player's sprite so that the baddies use exactly the same rules of movement but under computer control. What we do need, though, is a routine that adds a bit of intelligence to our baddies so that they will actually pursue our hero around the maze in true 'PacMan' style. Although such a routine may initially appear rather complicated, the best way to design a routine like this is to draw up a set of rules of behaviour that the baddies will follow.

Some games use a routine that makes the baddies pursue the hero no matter where he is within the maze, but this is hardly very realistic. If you think about it, if the maze was real and you were being hunted by a pack of drunken Millwall supporters (sorry lads!), they would probably be as lost as you are! The baddies should therefore only chase the player's sprite if they can actually see him. If they can't see him, they'll just continue searching around the maze until they find him. This obviously gives the player an advantage because they know where the baddies are, who are totally blind until the player comes into full view.

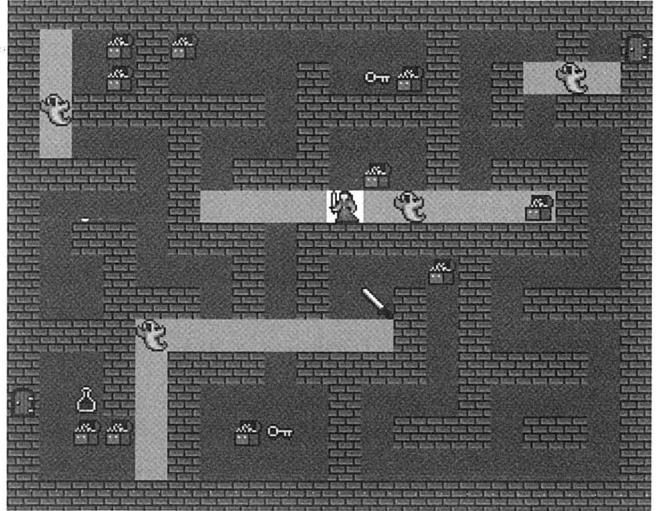
Bearing all this theory in mind, the baddie movement routine effectively works in two modes – Search mode (The baddies try to locate the player) and Pursue mode (they've located the player and immediately chase him). Search mode is pretty straightforward, but Pursue mode does need a little thought. The best way to handle this is to code a little routine that continuously checks all the clear blocks directly up, down, left and right of the baddie. If the routine encounters a wall in any one of these directions before it encounter the player's sprite then this direction is ignored. If the player's sprite is encountered, however, the baddie's direction of movement is immediately changed so that it starts to move in the direction of the player. This Pursue mode continues until the baddie either tracks down the player or the player manages to move out of the baddie's line of vision (if this happened, the baddie has effectively lost the player and will therefore start to search again). Here's the baddie movement code in psuedo code format.



Whenever the baddies are moved, the movement routine checks in all four directions to see whether the baddie can see the player.



### Checking 'lines of sight'



Start

Can baddie see player?

Yes

Move baddie in direction of player

No

Can baddie move left or right?

Yes

Can baddie move forward?

Yes

Does baddie want to move forward?

Yes

Move baddie forward

End If

No

Move baddie right or left

End If

No

Can baddie move forward?

Yes

Move baddie forward

No



```

        Can baddie move backwards?
        Yes
            Move baddie backwards
        End If
    End If
End If
End If
End of movement routine

```

With all the theory covered, let's take a look at a demonstration program that puts all this theory into practice. Although it's not yet a game in its own right, very little extra code needs to be added to get a very playable maze exploration game up and running. Here we go!

```

Rem *** Maze Game Demonstration
Rem *** Filename - MazeGameDemo.AMOS

```



```

Screen Open 0,320,256,32,Lowres
Flash Off : Curs Off : Cls 0

```

```

Rem *** Load icons...
Load "AMOSBOOK:AbkFiles/DungeonBlocks.ABK"
Get Icon Palette

```

```

Rem *** Load Bobs...
Load "AMOSBOOK:AbkFiles/DungeonSprites.ABK"

```

```

Rem *** Initialise Player Sprite Data structure...
Dim HERO(4)
HERO(0)=18 : Rem *** Map X location
HERO(1)=1 : Rem *** Map Y location
HERO(2)=2 : Rem *** Direction 1=North 2=South 3=West 4=East
HERO(3)=0 : Rem *** Movement step

```

```

Rem *** Initialise Ghost data structures
NUMGHOSTS=4
Dim GHOST(NUMGHOSTS,4)

```

```

For A=0 To NUMGHOSTS-1
  Read GHOST(A,0) : Rem *** Map X location
  Read GHOST(A,1) : Rem *** Map Y location
  Read GHOST(A,2) : Rem *** Direction
  Read GHOST(A,3) : Rem *** Movement step
Next A
Data 2,1,2,0,1,14,4,0,18,14,3,0,8,13,1,0

Rem *** Initialise Map data array...
Dim MAP(20,16)

Rem *** Initialise Bearing array...
Rem *** Holds directions of movement relative
Rem *** to player's position
Dim B(4,2)
For A=0 To 3
  Read B(A,0)
  Read B(A,1)
Next A
Data 0,-1,0,1,-1,0,1,0

Global MAP(),HERO(),B(),GHOST(),NUMGHOSTS,REDRAWFLAG

```

Before the player can start happily wandering around the demonstration maze, a fair bit of setting up has to take place. The program starts by opening up the game screen and then the sprite and icon banks are pulled into memory. Next, we start the serious business of building up the data structures for both the player and the ‘baddies’ that chase the player around the maze. Finally, a ‘bearing’ array is initialised that will tell the routines that handle both the movement of the player’s sprite and the baddies how to move around the maze. All these various data structures and variables are then made global so that all our procedures can have access to their contents.

```

Rem *** Point read pointer to level data...
Restore LEVEL1

```



```
Rem *** Draw Screen from map data...
Screen Hide 0
For Y=0 To 15
  For X=0 To 19
    Read BLOCK
    Paste Icon X*16,Y*16,BLOCK
    MAP(X,Y)=BLOCK
  Next X
Next Y
Screen Show 0
```

```
Double Buffer : Autoback 1
Bob Update Off : Hide
```

```
LEVEL1:
```

```
Data 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
Data 2,1,1,4,2,4,1,1,1,1,1,1,1,2,1,1,1,2,1,3
Data 2,1,1,4,2,1,1,1,1,2,1,6,4,2,1,2,1,1,1,2
Data 2,1,2,2,2,2,2,2,1,2,2,2,2,2,1,2,1,2,2,2
Data 2,1,1,1,1,2,1,1,1,2,1,1,1,2,1,1,1,1,1,2
Data 2,2,2,2,1,2,1,2,2,2,1,4,1,2,1,2,2,2,1,2
Data 2,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,4,2,1,2
Data 2,1,2,2,1,2,2,2,1,2,2,2,2,2,2,2,2,2,1,2
Data 2,1,1,2,1,1,1,2,1,2,1,1,1,4,2,1,1,1,1,2
Data 2,1,1,2,2,2,1,2,1,2,1,5,2,1,2,1,2,2,2,2
Data 2,2,2,2,1,1,1,1,1,1,1,1,1,2,1,2,1,1,1,2
Data 2,1,1,2,1,2,2,2,1,2,2,2,2,1,2,2,2,2,1,2
Data 3,1,7,2,1,2,1,1,1,1,2,1,1,1,2,1,1,1,1,2
Data 2,1,4,4,1,2,1,4,6,1,2,1,2,2,2,1,2,2,1,2
Data 2,1,1,1,1,2,1,1,1,1,2,1,1,1,1,1,1,1,1,2
Data 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
```

```
Repeat
```

```
  Bob Clear
```

```
Rem *** Remove object from new physical screen...
If REDRAWFLAG=1
```

```
    Paste Icon HERO(0)*16,HERO(1)*16,1
    REDRAWFLAG=0
End If

_MOVEHERO
_CHECKOBJECTS
_MOVEGHOSTS
_CHECKCOLLISIONS

Bob Draw
Screen Swap 0
Wait Vbl
Until MAP(HERO(0),HERO(1))=3
End
```

The screen is then drawn by pasting down 16 x 16 pixel icons that are held in the icon bank that we loaded at the start of the program. The code that handles this is virtually identical to the ‘Screen Icons’ demonstration listing that we covered in chapter 6. There’s one big difference here, however – instead of simply reading off and then discarding the data that is used to plot the icons, the data is placed into a dimensional array called ‘MAP()’ that is also used to control the movement of both the player and the baddies that chase him.

With the screen drawn, the program then enters the main game loop. The main game loop starts by checking that the variable ‘REDRAWFLAG’ has been set to 1. If it is, icon 1 is pasted down at the player’s current position. This is necessary because the AMOS command ‘Paste Icon’ doesn’t work properly with double buffered displays. If you were to simply paste an icon into the current logical screen, the physical screen would remain unchanged, causing a flicker between the new graphic that has been pasted into the logical screen and the graphic that was originally placed into the physical screen. If you don’t believe me, try removing these few lines of code!

Once this is done, the real action starts. The main game loop calls four procedures that are solely responsible for running the entire game. The first of these, `_MOVEHERO` (which we’ll be looking at next), updates

the position of the hero and this procedure is rapidly followed by the associated ‘\_CHECKOBJECTS’, ‘\_MOVEGHOSTS’ and ‘\_CHECKCOLLISIONS’ procedures that check when the player has walked over an object, updates the positions of the ghosts (this is the most complicated routine of the whole game!) and checks whether the ghosts have run into the player’s sprite. Anyway, let’s take a look at the ‘\_MOVEHERO’ procedure.

#### Procedure \_MOVEHERO

```
BRNG=0
```

```
Rem *** Has the player stopped moving?
```

```
If HERO(3)=0
```

```
    Rem *** Check joystick position...
```

```
    If Joy(1) and 1
```

```
        BRNG=1 : Rem *** North
```

```
    End If
```

```
    If Joy(1) and 2
```

```
        BRNG=2 : Rem *** South
```

```
    End If
```

```
    If Joy(1) and 4
```

```
        BRNG=3 : Rem *** West
```

```
    End If
```

```
    If Joy(1) and 8
```

```
        BRNG=4 : Rem *** East
```

```
    End If
```

```
If BRNG<>0
```

```
    Rem *** Can the player move in that direction?
```

```
    If MAP(HERO(0)+B(BRNG-1,0),HERO(1)+B(BRNG-1,1))<>2
```

```
        Rem *** Start to move hero...
```

```
        HERO(2)=BRNG
```

```
        HERO(3)=1
```

```
    End If
```

```
End If
```



```

Else
  Rem *** Has the hero moved all the way
  Rem *** to the next square?
  If HERO(3)=7
    BRNG=HERO(2)

    Rem *** Update location of hero...
    HERO(0)=HERO(0)+B(BRNG-1,0)
    HERO(1)=HERO(1)+B(BRNG-1,1)

    Rem *** Reset movement step...
    HERO(3)=0
  Else
    Rem *** Increment movement step...
    HERO(3)=HERO(3)+1
  End If
End If
_DRAWHERO
End Proc

```

It may look rather long, but there's nothing particularly complex about this procedure. All it does is to handle the movement of the player's sprite around the maze by making sure that the player doesn't manage to walk through walls. The player's sprite has a status variable attached to it that tells AMOS when it has managed to walk from one square to another. The `_MOVEHERO` routine therefore starts by checking to see whether the player's sprite is already walking in a particular direction or whether it has reached the grid it is walking to (which means that the sprite has effectively stopped). If the player's sprite has stopped moving, the joystick is checked and the 'BRNG' variable (short for 'Bearing') is set to a value between 1 and 4. A value of 1 means that the player has tried to move up, a value of 2 means down, 3 means left and 4 means right.

The movement routine then checks the map data array to see whether the player can actually move in that direction. The direction of movement is taken from the 'B()' array that contains bearing values that offset the player's current position by the correct number of units. If the player

wishes to move north, for example, the values '-1' and '0' are temporarily added to the player's current position, effectively checking the block immediately above the player. If the player can move in that direction (i.e. the block doesn't contain a value of '2' – indicating a wall), then he is moved accordingly. If he can't, however, the joystick input is simply ignored.

If the player's sprite has already started moving to a new position, the routine then checks to see whether the next movement will complete the step from one block to another. If it will, the player's position within the map is set and the status variable is reset to zero, otherwise the status variable is increased. This status value will be used to calculate the exact pixel position of the player's sprite relative to his position within the map.

#### Procedure \_DRAWHERO

```

FRAME=1

Rem *** Is the hero moving up or down?
If HERO(2)<3
    HEROX=HERO(0)*16

    Rem *** Is the hero moving down?
    If HERO(2)=1
        HEROY=HERO(1)*16-(HERO(3)*2)
    Else
        HEROY=HERO(1)*16+(HERO(3)*2)
    End If
Else
    Rem *** Is the hero moving left?
    If HERO(2)=3
        HEROX=HERO(0)*16-(HERO(3)*2)
        FRAME=2
    Else
        HEROX=HERO(0)*16+(HERO(3)*2)
    End If
    HEROY=HERO(1)*16

```





```

End If

Rem *** Draw Hero Bob...
Bob 0,HEROX,HEROY,FRAME
End Proc

```

The ‘\_DRAWHERO’ routine is – not surprisingly – responsible for drawing the player’s sprite onto the screen. Because the player’s sprite can be facing either left or right, the image that will be drawn onto the screen is calculated by checking the direction that the player is moving in. Not only that, but the value held in the hero’s status variable could be added to either its ‘X’ or ‘Y’ co-ordinate. The routine therefore checks which direction the player is moving in and adds the status value to the appropriate co-ordinate. If the player was moving north, for example, then the step value is subtracted from the player’s ‘Y’ co-ordinate and if they are moving east, it’s added to the sprite’s ‘X’ co-ordinate. Finally, the player’s sprite (well, it’s a bob actually) is drawn onto the screen.

#### Procedure \_CHECKOBJECTS



```

Rem *** Is the player standing over an object?
If MAP(HERO(0),HERO(1))>3
    Rem *** Remove object from screen map...
    MAP(HERO(0),HERO(1))=1
    Bell

    Rem *** Clear object graphic...
    Paste Icon HERO(0)*16,HERO(1)*16,1

    Rem *** Set Redraw flag so that icon is drawn into
    Rem *** Both the physical and logical screens...
    REDRAWFLAG=1
End If
End Proc

```

The ‘\_CHECKOBJECTS’ procedure is used to check whether the player’s sprite has walked over one of the many objects scattered throughout the maze. The routine works by checking the value held in

the 'MAP()' array at the same position as the player's sprite to see whether it is greater than 3 (indicating an item of treasure). If an object is found, the object is effectively 'picked up' by the player and removed from the screen by pasting down icon 1 and changing the value held at that position in the 'MAP()' array to 1 (indicating a basic floor square).

#### Procedure \_MOVEGHOSTS

```
FOUND=-2
```

```
Rem *** Update each ghost in turn...
```

```
For A=0 To NUMGHOSTS-1
```

```
Rem *** Has ghost stopped moving?
```

```
If GHOST(A,3)=0
```

```
Rem *** Can ghost see player?
```

```
_SEARCHFORHERO[A]
```

```
FOUND=Param
```

```
Rem *** If so, move ghost...
```

```
If FOUND>-1
```

```
    GHOST(A,2)=FOUND
```

```
    GHOST(A,3)=1
```

```
End If
```

```
If FOUND=-1
```

```
Rem *** Can ghost move left or right?
```

```
_CHECKLEFTRIGHT[A]
```

```
FOUND=Param
```

```
If FOUND>-1
```

```
    BRNG=GHOST(A,2)
```

```
    MAPX=GHOST(A,0)+B(BRNG-1,0)
```

```
    MAPY=GHOST(A,1)+B(BRNG-1,1)
```

```
Rem *** Does ghost have the option
```

```
Rem *** of going Forward?
```



```

If MAP(MAPX,MAPY)=2 or MAP(MAPX,MAPY)=3

    Rem *** If not, then go left or right
    GHOST(A,2)=FOUND
    GHOST(A,3)=1
Else
    Rem *** Does ghost want to change
    Rem *** direction?
    DIRECTION=Rnd(2)
    If DIRECTION=1
        Rem *** If so, then change
        Rem *** direction
        GHOST(A,2)=FOUND
        GHOST(A,3)=1
    Else
        FOUND=-1
    End If
End If
End If
End If

If FOUND=-1
    Rem *** Can ghost go forward?
    _CHECKFORWARD[A]
    FOUND=Param
    If FOUND=1
        Rem *** If so, go forward...
        GHOST(A,3)=1
    Else
        Rem *** Can ghost go backwards?
        _CHECKBACKWARDS[A]
        FOUND=Param
        If FOUND>-1
            Rem *** If so, go backwards...
            GHOST(A,2)=FOUND
            GHOST(A,3)=1
        End If
    End If
End If

```

```

        End If
    End If
Else
    Rem *** Has the ghost moved all the way
    Rem *** to the next square?
    If GHOST(A,3)=7
        BRNG=GHOST(A,2)

        Rem *** Update location of ghost...
        GHOST(A,0)=GHOST(A,0)+B(BRNG-1,0)
        GHOST(A,1)=GHOST(A,1)+B(BRNG-1,1)

        Rem *** Reset movement step...
        GHOST(A,3)=0
    Else
        Rem *** Increment movement step...
        GHOST(A,3)=GHOST(A,3)+1
    End If
End If
Next A

Rem *** Redraw all the ghost bobs
_DRAWGHOSTS
End Proc

```

The ‘\_MOVEGHOSTS’ procedure and the routines that it calls are the most complex aspect of our demonstration game and they closely follow the pseudo code that we covered earlier. The routine processes all four ghosts using a loop that counts from 0 to 3. Just like the hero movement routine, it starts by checking whether the ghost has completed a single movement from one step to another. If it has, the routine calls a procedure called ‘\_SEARCHFORHERO’ that checks to see whether a particular ghost can actually see the hero. If the value returned by that routine is greater than ‘-1’, the baddie is moved in the direction of the player, effectively making the ghost ‘chase’ the player.

If the baddie can’t see the player, however, it then checks to see whether the baddie can move left or right by calling the ‘\_CHECKLEFTRIGHT’

procedure. If the ghost can, the routine then checks to see whether the baddie has a choice (it might, for example, have reached the end of a corridor and is staring at a brick wall!). If the baddie doesn't have a choice, then he changes direction. If the baddie does have the choice, however, a random number is generated between one and zero. If a value of one is returned, then the baddie changes direction, otherwise the routine carries on.

Finally, the routine checks to see if the baddie can go forward. If the baddie can go forward, then the move is made. If not, the routine then checks to see if it can move backwards by calling the '\_CHECKBACKWARDS' procedure.

The rest of the code is almost identical to the second half of the '\_MOVEHERO' procedure that we covered earlier. All it does is to move the ghost from one block to another by incrementing the value of the baddie's 'status' variable ('GHOST(n,3)').

```
Procedure __DRAWGHOSTS
```

```
    FRAME=3
```

```
    For A=0 To NUMGHOSTS-1
```

```
        If GHOST(A,2)<3
```

```
            GHOSTX=GHOST(A,0)*16
```

```
            Rem *** Is the ghost moving down?
```

```
            If GHOST(A,2)=1
```

```
                GHOSTY=GHOST(A,1)*16-GHOST(A,3)*2
```

```
            Else
```

```
                GHOSTY=GHOST(A,1)*16+GHOST(A,3)*2
```

```
            End If
```

```
        Else
```

```
            Rem *** Is the ghost moving left?
```

```
            If GHOST(A,2)=3
```

```
                GHOSTX=GHOST(A,0)*16-GHOST(A,3)*2
```

```
                FRAME=4
```



```

    Else
        GHOSTX=GHOST(A,0)*16+GHOST(A,3)*2
    End If
    GHOSTY=GHOST(A,1)*16
End If

Rem *** Draw Ghost Bob...
Bob A+1,GHOSTX,GHOSTY,FRAME
Next A
End Proc

```

The ‘\_DRAWGHOSTS’ procedure is almost identical to the ‘\_DRAWHERO’ procedure. For more information, refer back to that procedure.

```
Procedure _SEARCHFORHERO[GHOST]
```

```

BLOCK=0
FOUND=-1

Rem *** Is the ghost standing on a different square
Rem *** from the player?
If GHOST(GHOST,0)<>HERO(0) or GHOST(GHOST,1)<>HERO(1)
    For C=0 To 3
        MAPX=GHOST(GHOST,0)
        MAPY=GHOST(GHOST,1)

        XINC=B(C,0)
        YINC=B(C,1)
        WALL=0

    Repeat
        Rem *** Can player be seen?
        If MAPX=HERO(0) and MAPY=HERO(1)
            FOUND=C+1
        End If
        Rem *** Is a wall block ghost's view?

```



```

    If MAP(MAPX,MAPY)=2 or MAP(MAPX,MAPY)=3
        WALL=1
    End If

    If WALL=0
        Rem *** Look a square further...
        MAPX=MAPX+XINC
        MAPY=MAPY+YINC
    End If
Until FOUND>-1 or WALL=1

Next C
End If
End Proc[FOUND]

```

The ‘\_SEARCHFORHERO’ procedure checks to see whether the baddie can see the player’s sprite in any one of the four directions of movement. The routine starts by checking to see whether the baddie is standing on a different square from the player’s sprite. After all, there’s little point in searching for the player if the ghost has already located him.

Each of the four directions of movement are checked by starting from the baddie’s current block position and then moving steadily out until the baddie’s line of vision is either obstructed by a wall or the baddie claps his eyes on the player’s sprite. A very clever routine, I’m sure you’ll agree!

```

Procedure _CHECKLEFTRIGHT[GHOST]
    FOUND=-1
    BRNG=GHOST(GHOST,2)
    MAPX=GHOST(GHOST,0)
    MAPY=GHOST(GHOST,1)

    Rem *** Is the ghost facing north or south?
    If BRNG<3
        Rem *** Can ghost turn left?

```



```
    If MAP(MAPX-1,MAPY)<2 or MAP(MAPX-1,MAPY)>3
        FOUND1=3
    End If
    Rem *** Can ghost turn right?
    If MAP(MAPX+1,MAPY)<2 or MAP(MAPX+1,MAPY)>3
        FOUND2=4
    End If
    Rem *** Can ghost go left and right?
    If FOUND1=3 and FOUND2=4
        Rem *** Which direction does ghost
        Rem *** want to go?
        CHOOSDIR=Rnd(2)
        If CHOOSDIR=0
            Rem *** Go left
            FOUND=FOUND1
        Else
            Rem *** Go right
            FOUND=FOUND2
        End If
    Else
        Rem *** Can't ghost go in either direction?
        If FOUND1=0 and FOUND2=0
            FOUND=-1
        Else
            Rem *** Which direction can ghost go?
            If FOUND1<>0
                FOUND=FOUND1
            Else
                FOUND=FOUND2
            End If
        End If
    End If
Else
    Rem *** This code is executed if ghost is
    Rem *** facing left or right...
    Rem *** It's virtually the same as the code above
```



```
    If MAP(MAPX,MAPY-1)<2 or MAP(MAPX,MAPY-1)>3
        FOUND1=1
    End If
    If MAP(MAPX,MAPY+1)<2 or MAP(MAPX,MAPY+1)>3
        FOUND2=2
    End If
    If FOUND1=1 and FOUND2=2
        CHOOSDIR=Rnd(2)
        If CHOOSDIR=0
            FOUND=FOUND1
        Else
            FOUND=FOUND2
        End If
    Else
        If FOUND1=0 and FOUND2=0
            FOUND=-1
        Else
            If FOUND1<>0
                FOUND=FOUND1
            Else
                FOUND=FOUND2
            End If
        End If
    End If
End Proc[FOUND]
```

The ‘\_CHECKLEFTRIGHT’ procedure contains two almost identical sections of code that check the positions directly left and right of the baddie’s current position relative to the direction it is facing. The routine first checks to see whether the baddie can move left and then right. If the baddie can move in both directions, then a decision must be made – which direction is the baddie going to choose? Left or right? This is decided by generating a random number between 0 and 1. If, on the other hand, the baddie can’t move in either direction, then a value of ‘-1’ is returned. Otherwise, the direction of movement (a bearing value) is returned.

```

Procedure _CHECKFORWARD[GHOST]
  FOUND=-1
  BRNG=GHOST(GHOST,2)
  MAPX=GHOST(GHOST,0)+B(BRNG-1,0)
  MAPY=GHOST(GHOST,1)+B(BRNG-1,1)

  Rem *** Can ghost go forward?
  If MAP(MAPX,MAPY)<2 or MAP(MAPX,MAPY)>3
    FOUND=1
  End If
End Proc[FOUND]

```



This very simple procedure checks to see whether the baddie can move directly forward. Obviously this is directly affected by the baddie's current direction – if the baddie is facing left, for example, then forward will be left. If the baddie is facing down, on the other hand, then forward will be down. The routine simply checks the block that is immediately next to the current position of the baddie in the 'MAP()' array. If any value other than either '2' or '3' is found, then the movement is allowed.

```

Procedure _CHECKBACKWARDS[GHOST]
  FOUND=-1
  BRNG=GHOST(GHOST,2)

  Rem *** Reverse direction of movement
  If BRNG<3
    If BRNG=1
      BRNG=2
    Else
      BRNG=1
    End If
  Else
    If BRNG=3
      BRNG=4
    Else
      BRNG=3
    End If
  End If

```



```

End If

MAPX=GHOST(GHOST,0)+B(BRNG-1,0)
MAPY=GHOST(GHOST,1)+B(BRNG-1,1)

Rem *** Can ghost go backwards?
If MAP(MAPX,MAPY)<2 or MAP(MAPX,MAPY)>3
    FOUND=BRNG
End If
End Proc[FOUND]

```

The baddie movement routine is completed with the addition of the ‘\_CHECKBACKWARDS’ procedure that checks to see whether, after all other movement checks have been made, the baddie can move backwards. Once again, moving backwards is directly relative to the baddie’s current direction – if the baddie is facing left, then moving the baddie backwards would make it move right. The routine starts by reversing the baddie’s ‘bearing’ and once this bearing value has been calculated, the ‘MAP()’ array is checked. If any value other than ‘2’ or ‘3’ is found, the movement is legal and the bearing value is returned.

```

Procedure _CHECKCOLLISIONS
    Rem *** Has hero collided with any ghosts?
    STATUS=Bob Col(0,1 To NUMGHOSTS)
    If STATUS=-1
        Rem *** Insert your collision code here!....
        Bell
    End If
End Proc

```



Phew! – the last procedure. This is actually a fairly minimal procedure that I added just to show you how to add collision detection to such a game. At the moment, the procedure simply plays the AMOS ‘Bell’ sound if a collision takes place between the player’s sprite and the baddies, but you could quite easily add your own code. If you’re feeling adventurous, why not add a combat routine so that the player has a chance of killing the baddies when the fire button is pressed? Happy coding!

## 3D 'Dungeon Master' games

We have covered some pretty heavy games programming tricks and techniques over the past few pages. If you've managed to master the maze code, then you'll be ready to transform that code into the next dimension. Programming 3D exploration games such as FTL's 'Dungeon Master' or SSI's brilliant 'Eye of the Beholder II' may seem an almost impossible task, but you'll be surprised to learn that they're actually very simple indeed. OK, so they look damned impressive, but the code required to generate those 3D displays is actually very straightforward.

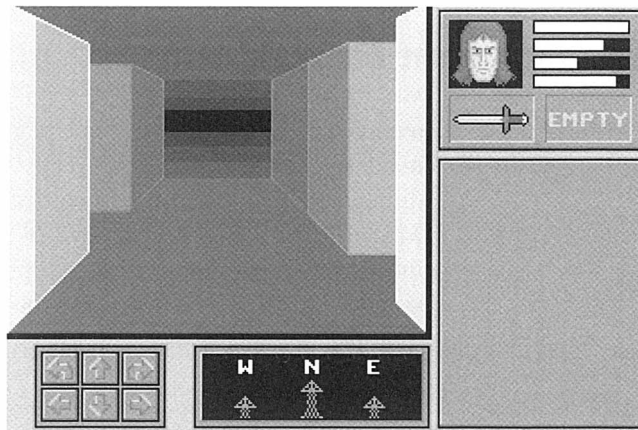
Don't worry if the thought of 3D graphics reduces you to a cold sweat – believe it or not, but there is virtually no 3D mathematics involved at all. What's more, you don't even need AMOS 3D to write a 'Dungeon Master' clone – all you need to get started is a copy of AMOS and Deluxe Paint. First, though, let's explain how we get the maze code to work in 3D.

Although games like 'Dungeon Master' look very complex, if you study them very closely you will notice that the 3D display is actually very simple. All the walls, walkways and doors you'll encounter in these games are actually made up of discreet graphics blocks that are pasted down in the correct positions to build up a 3D display. The actual 3D map data is held as a dimensional array – in fact, it's exactly the same data that is used to draw the conventional maze display we covered earlier!

So how do you draw the 3D display? Well, all you do is to draw up a set of icons which represent each block as it would be viewed in solid 3D. This invariably takes time to get right – probably the best approach is to draw up a 3D grid similar to the one shown elsewhere on this page. You can then cut out each section and draw it in as a solid object. All these wall parts are then saved to disk, cut out within the AMOS Object Editor and placed into an Icon bank.

To draw the 3D grid, you just interrogate the map data array to find out which blocks are empty, which blocks contain walls and which blocks contain doors. Doors are a strange case, however – while they are

*Fancy writing your own  
Dungeon Master clone?  
Well now you can! What's  
more, it's not as difficult  
as you might think.*



technically the same as walkways (in that you can walk through them), the routine that constructs the display draws them first a solid walls and then pastes the door graphics on top of the wall icons, therefore creating the illusion of a door. Of course it would be ridiculous to draw every block within the map data array (after all, a 100 x 100 map data array would contain 10,000 possible blocks!), so you need to draw only those blocks that are actually visible.

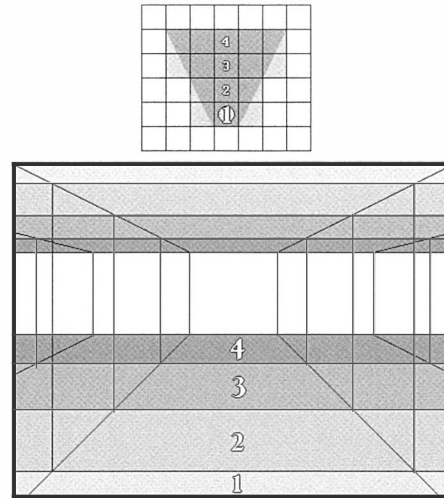
As you can see from the diagram, only fifteen blocks are actually drawn, starting from the back and then working forward – five at the back, then another five, another three and then finally two at either side of the player's viewpoint. Not all of these blocks will be directly visible – some will be hidden by other blocks appearing nearer to the player's position. If an Icon is drawn over the top of another Icon, that's the way it goes.

You could speed up your 3D drawing routine by writing a piece of code that draws only those blocks that are actually visible, but the code required to do this tends to slow the game down too much. I've managed to get a very playable 'Dungeon Master' clone up and running using the technique covered in the last paragraph and I can assure you that it is very quick indeed – because AMOS uses the Amiga blitter chip to plot these Icons, it is capable of drawing all 15 wall blocks in just 4 frames (just under a tenth of a second). As any games programmer will tell you, that's more than enough for the most demanding 'Dungeon Master' clone.



**Drawing  
'visible'  
blocks only**

The data held in the map array is transformed into 3D by reading off the data in strips and then plotting the correct icons onto the screen.



Drawing the 3D display, however, is only half the battle. Unlike the simple maze game that we covered earlier, the player doesn't just move up, down, left and right. Just like the real world, a 3D game must handle directions relative to the player's bearing – that is, the direction the player is facing. To keep things nice and simple, all we need are four bearings – North, South, East and West.

As you can probably appreciate, taking account of the player's bearing introduces another problem. To demonstrate this problem, consider the following – will a player that is facing north move in the same direction as player that is facing south when they move forward? Of course not! – Because the action of moving forward is entirely relative to the direction that you're facing, a move forward in a southerly direction will produce exactly the opposite effect to a movement forward in a northerly direction. To handle this you need to create another dimensional array to hold another set of co-ordinates that control the direction of movement relative to the player's bearing. For example, to move a player facing north forward, the 'Y' (up and down) co-ordinate would be decremented whilst the 'X' (left and right) co-ordinate would remain unchanged. To carry out the same operation in a southerly direction, the 'Y' co-ordinate would be incremented, therefore creating the opposite (but technically correct) effect.



# Platform games

- Drawing platforms
- Tying 'baddies' to platforms
- Jumping between platforms
- Picking up objects



One of the most popular game genres these days is the humble platform game, which first saw light in the early days of 8-bit micros like the Spectrum and Commodore 64. Games like ‘Manic Miner’ and ‘Jet Set Willy’ were nothing special in the graphics or sound department, but the combination of jumping up and down platforms, picking up objects whilst avoiding nasties, gave them that all-important spark of playability that ensured the popularity of platform games for years to come. Even in these days of CD32s, 64-bit Nintendos and the Sanyo 3D0 games console, platform games still thrive. Indeed, platform games are so popular that both Sega and Nintendo use characters from two of the most popular platform games of all time as their mascots – Sonic the Hedgehog and Super Mario. Both these games may be a far cry from the likes of ‘Jet Set Willy’, but the gameplay is the same!

Writing platform games may seem very difficult indeed, but you’ll be surprised just how easy it really is. The best way to understand just about any type of game is to sit down and consider exactly what elements the game contains. In the case of a platform game, you have essentially just four different elements to the game.

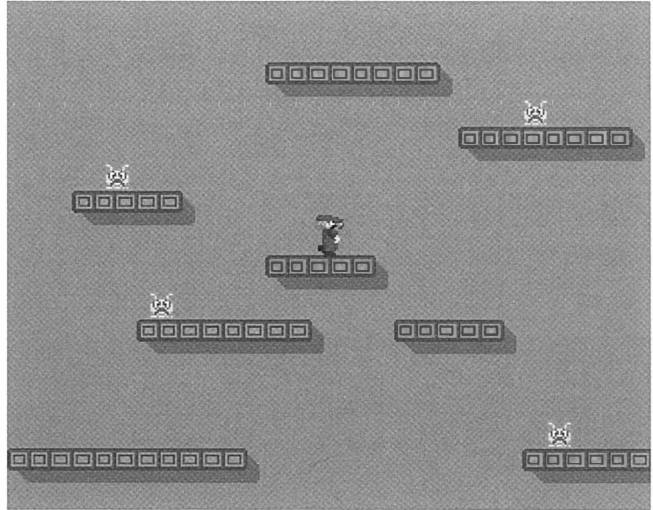
**Platforms** Pretty obvious this. Platforms are the flat levels that the player’s sprite can walk along, fall and jump on to.

**The player’s sprite** The player’s sprite must be able to walk left and right, jump up onto other platforms and fall down gaps between platforms. We’ve already looked at how to make a sprite ‘jump’ in chapter 8, so the rest is quite simple.

**Baddies** All platform games ranging from ‘Impossible Mission’ (remember that one?) to ‘Super Mario Land’ have some form of baddie that roams around on certain platforms to make the game a little more challenging. Without these baddies, the player would be able to jump between platforms with ease!

**Collectables** No, we’re not trying to write the platform game equivalent of the ‘Antiques Road Show’! Collectables are objects that the player must collect in order to either complete each screen or to increase their score.

*Rainbow Islands it may not be, but with a little bit of imaginative coding there's no reason why our AMOS platform game couldn't rival the best of 'em!*



### **Building platform screens**

Let's start by taking a look at how the platforms are handled. Most budding games programmers make the mistake of designing their game screens using a paint program such as DPaint. Whilst this approach will work, a much better solution is to use screen icons to build up the display from an array of numbers. If you skimmed through chapter 6, then you will have missed out on the programming technique required for this sort of thing, so why not take a quick look back to see exactly what you missed?

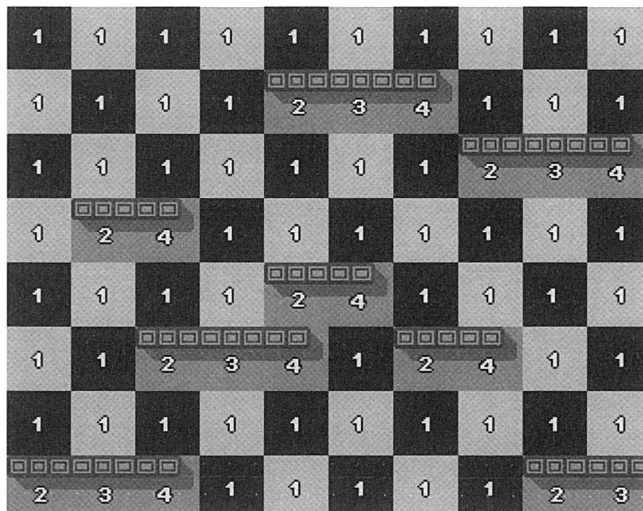


### **'Screen zones' facility**

Drawing up a screen display filled with platforms is all well and good, but how to you get AMOS to recognise them? Once again, AMOS has the answer thanks to its very useful 'Screen Zones' facility that we covered back in chapter 6. All you do is to define the areas of the screens that contain the platforms as screen zones. The most obvious way of doing this is to work out the screen co-ordinates for each platform yourself and then enter them into your program manually as data statements. A much better way, however, is to let AMOS do the work.

The demonstration program that accompanies this section on platform games uses this technique to great effect – what's more, because the positions of the platform screen zones are calculated from the data used to draw the screen up using screen icons, you can rearrange your platforms and even add new platforms without ever having to worry

The positions of the platforms are automatically calculated by interpreting the same data that is used to draw up the screen display.



about defining new screen zones. I have to admit that I'm pretty chuffed with this routine – every single attempt at an AMOS platform game that I've ever come in the past as always relied on the programmer to calculate the positions of the platform screen zones themselves, so my routine is very, very simple to use.

You can add hundreds of different screen layouts as you like and my routine will automatically handle the different platform positions automatically!

The player's sprite interacts with the platforms using a very handy function called 'Zone()' that checks to see whether there is a screen zone at a given screen position (in the case of a platform game, the position of the player's sprite). If a zone is found, then the game assumes that the player's sprite is standing on a platform. If a value of zero is returned however (meaning that there isn't a hot spot at this position), then the player's sprite is made to fall down the screen in a straight line until the sprite either drops off the bottom of the screen (in which case you can choose to either kill the player's sprite, or make it wrap around to the top of the screen in true 'Bubble Bobble' fashion) or it lands on a platform below.

## Adding baddies

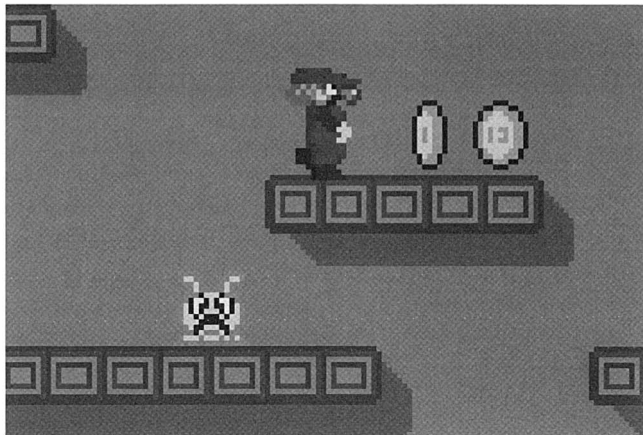
Adding baddies to your platform game is surprisingly simple too. The best way that I've found is to create a data structure for each baddie (aren't data structures wonderful!) that contains the screen co-ordinates of the baddie, the baddie's speed and direction of movement, the number of the platform that the baddie is walking on and the number of the image in the sprite bank that is to be used for that baddie. You'll also need to create a dimensional array that contains the screen co-ordinates of each and every platform that you use. This array is automatically filled with the appropriate values as the screen zones are initialised.

It's not even necessary to calculate the exact screen position of your baddies, either. Because you've stored the co-ordinates of the platform screen zones in an array, you can simply extract the necessary values from that array by checking what platform the baddie is to stand on. The number of the platform is then used to extract the left 'X' and top 'Y' co-ordinates of the platform from the array containing the co-ordinates of its screen zone, which are then placed into the 'X' and 'Y' co-ordinates of the baddie. Moving the baddie backwards and forwards along the platform is just as easy too – just add the value held in the baddie's speed variable to its 'X' screen position and then check that the baddie hasn't walked off either end of the platform by comparing the baddie's 'X' screen position to the left and right 'X' co-ordinates of the platform that the baddie is standing on. Simple, eh?



**Working from platform positions**

*Adding 'baddies' and objects for your AMOS platform explorer to pick up is surprisingly simple.*



## Collectables

Collectables are handled pretty much in the same way as the baddies, the only difference being that the collectables don't actually move. As a result, all you need to do is to create a data structure that contains the number of the platform that the collectible is on, a positive offset value that places the object to the right of the left hand edge of the platform, the object's status (has it already been picked up? If so, don't draw the object bob again) and the number of points (or whatever) that the player earns when they collect it. You can then calculate the exact screen position of the object using exactly the same technique that we used for the baddies. It's as simple as that!

If you manage to grasp all these concepts, then there's absolutely no reason why you couldn't write your own platform games along the same lines as 'Manic Miner' and 'Impossible Mission'. Let's take a look at a demonstration program that illustrates the points that we've covered. In order to make the listing as understandable as possible, however, I've broken it down into small chunks which are accompanied by a short description of how what that section of code does and how it works.

For the sake of argument we'll call our hero 'Mario', but please note that this is *not* a copy of a Mario game, and Nintendo would get very cross if you wrote one and tried to distribute it!

```
Rem *** Platform Game Demonstration
Rem *** Filename - PlatformDemo.AMOS
```



```
Screen Open 0,320,256,16,Lowres
Flash Off : Curs Off : Cls 0 : Hide
```

```
Load "AMOSBOOK:AbkFiles/PlatformIcons.abk"
Load "AMOSBOOK:AbkFiles/Mario.abk"
Get Sprite Palette
```

```
Rem *** Define Mario Bob data structure
```

```
Dim MARIO(4)
MARIO(0)=32 : Rem *** Bob X Position
MARIO(1)=224 : Rem *** Bob Y Position
MARIO(2)=0 : Rem *** Status 0=Walk 1=Jump 2=Fall
MARIO(3)=0 : Rem *** Jump angle
```

```
Rem *** Define Baddie Bob data structure...
NUMBADDIES=4
```

```
Dim BADDIE(NUMBADDIES,5)
BADDIE(0,0)=0 : Rem *** Baddie X Position
BADDIE(0,1)=0 : Rem *** Baddie Y Position
BADDIE(0,2)=2 : Rem *** Baddie Speed and direction
BADDIE(0,3)=4 : Rem *** Platform that baddie is on
BADDIE(0,4)=7 : Rem *** Baddie bob image number
```

```
Rem *** Baddie 2...
BADDIE(1,2)=1
BADDIE(1,3)=2
BADDIE(1,4)=7
```

```
Rem *** Baddie 3...
BADDIE(2,2)=1
BADDIE(2,3)=7
BADDIE(2,4)=7
```

```
Rem *** Baddie 4...
BADDIE(3,2)=3
BADDIE(3,3)=1
BADDIE(3,4)=7
```

```
Rem *** Set hot spot of Mario and Baddie Bobs
For C=1 To 7
    Hot Spot C,$12
Next C
```

```
Rem *** Platform() array holds 'Y' position of
```

```
Rem *** top of platform
Dim PLATFORM(20,4)

FRAME=1 : FRAMEDELAY=0 : SPEED=4
Global DIRECTION,XOFFSET,YOFFSET,MARIO()
Global SPEED,FRAME,PLATFORM(),NUMBADDIES,BADDIE()

_SETUPSCREEN
```

Before the player can start jumping between platforms, quite a bit of setting up is necessary. After opening the game screen and loading the game icons and sprites, several data structures need to be created. The first, 'MARIO()', holds information on the player's sprite. Four data items are attached to the sprite – it's current 'X' and 'Y' screen position, the angle value which is used by the '\_JUMP' procedure and a status variable that can contain one of three different values which tells the program what the sprite is doing. If it contains a value of one, for example, then the sprite is jumping between one platform and another.

The baddies that inhabit the platforms that are scattered around the screen each have a data structure assigned to them too. Called 'BADDIE()', the data structure holds such information as the baddie's current 'X' and 'Y' screen position, it's speed and direction (the baddie's direction is indicated by a positive or negative value), the number of the platform that the baddie is standing on and the baddie's Sprite bank image number.

The 'Y' screen position of the top of each platform is held in an array called 'PLATFORM()'. This is used not only to correctly position the baddies, but also to place the sprite when it either jumps or falls onto a platform. Finally, all this data is made global so all the game procedures have access to it and the screen is set up by calling the imaginatively named procedure '\_SETUPSCREEN'.

Do



Bob Clear

```

_CHECKCKJOYSTICK
_CHECKKFALL
_CHECKKLANDED
_MOVEBADDIE
_CHECKKCOLLISIONS

```

Bob 0, MARIO(0), MARIO(1), FRAME

For A=0 To NUMBADDIES-1

Bob A+1, BADDIE(A, 0), BADDIE(A, 1), BADDIE(A, 4)

Next A

Bob Draw

Screen Swap 0

Wait Vbl

Rem \*\*\* Update Mario animation

If FRAMEDELAY&gt;6-SPEED

FRAMEDELAY=0

FRAME=FRAME+1

If FRAME=7 or FRAME=\$8007

FRAME=FRAME and \$8000

FRAME=FRAME+1

End If

End If

FRAMEDELAY=FRAMEDELAY+1

Loop

The main game loop is fairly straightforward. Because we've turned off automatic bob redraws, the loop starts by removing all bobs from the screen. Five procedures are then called that form the backbone of the game. The first, '\_CHECKJOYSTICK' reads the joystick, '\_CHECKKFALL' checks to see whether the player is falling between platforms, '\_CHECKKLANDED' checks to see whether the player has landed on a platform, '\_MOVEBADDIE' updates the positions of all



baddie objects and ‘\_CHECKCOLLISIONS’ is responsible for collision detection.

Once all five of these procedures have done their stuff, both the ‘Mario’ bob and the baddie bobs are drawn onto the screen. The player’s sprite uses quite a complex animation so this is the last thing to be updated by the main game loop.



#### Procedure \_SETUPSCREEN

```

PLATFORM=0 : Rem *** Current platform number
PLATFLAG=0 : Rem *** 1 = Platform start 2 = Platform ended

Rem *** Draw screen display using icons
Restore LEVEL1DATA

Rem *** No more than 20 platforms per screen!
Reserve Zone 20

For Y=0 To 7
  For X=0 To 9
    Read BLOCKNUM
    Paste Icon X*32,Y*32,BLOCKNUM

Rem *** Calculate start position of zone
If BLOCKNUM=2
  X1=X*32
  Y1=Y*32
  PLATFLAG=1
End If

Rem *** Calculate end position of zone
Rem *** and initialise it
If BLOCKNUM=4
  X2=(X*32)+23
  Y2=(Y*32)+11
  Set Zone PLATFORM+1,X1,Y1 To X2,Y2

```

```
PLATFORM(PLATFORM,0)=X1
PLATFORM(PLATFORM,1)=Y1
PLATFORM(PLATFORM,2)=X2
PLATFORM(PLATFORM,3)=Y2

PLATFORM=PLATFORM+1
PLATFLAG=0
End If

Rem *** Has the end of the screen been reached
Rem *** and the end of a platform has not been found?
If X=9 and PLATFLAG=1
    X2=(X*32)+32
    Y2=(Y*32)+11
    Set Zone PLATFORM+1,X1,Y1 To X2,Y2

    PLATFORM(PLATFORM,0)=X1
    PLATFORM(PLATFORM,1)=Y1
    PLATFORM(PLATFORM,2)=X2
    PLATFORM(PLATFORM,3)=Y2

    PLATFORM=PLATFORM+1
    PLATFLAG=0
End If
Next X
Next Y

Rem *** Screen icon map data
LEVEL1DATA:
Data 1,1,1,1,1,1,1,1,1,1
Data 1,1,1,1,2,3,4,1,1,1
Data 1,1,1,1,1,1,1,2,3,4
Data 1,2,4,1,1,1,1,1,1,1
Data 1,1,1,1,2,4,1,1,1,1
Data 1,1,2,3,4,1,2,4,1,1
Data 1,1,1,1,1,1,1,1,1,1
Data 2,3,3,4,1,1,1,1,2,3
```

```

Bob Update Off
Double Buffer : Autoback 1
End Proc

```

The ‘\_SETUPSCREEN’ procedure is a very clever routine that not only draws up the screen display using icons stored in the icon bank that we loaded at the start of the program, but also calculates the positions of the platforms based on the same data that is used to plot the icons. Because the icons are stored as screen zones (refer back to chapter 6 for more on screen zones), the routine starts by reserving enough memory to handle up to 20 zones. Our demo game doesn’t use even half this amount, but it’s always wise to reserve enough memory for later levels that may possibly be more complex.

With the memory for our screen zones reserved, the routine then enters two loops that are responsible for reading in the screen icon data, plotting the appropriate icons onto the screen and then calculating the positions of each and every platform according to this data. I have to admit that I’m quite pleased with this particular aspect of the program – even if you change the level data drastically, the platforms will still be handled correctly. So how does it work? Well, the position of each platform is calculated according to strict rules – each platform must start with icon two (a graphic of the beginning of a platform) and end with icon four (a graphic of the end of a platform). There is an extra platform graphic that can be used to extend a platform (icon three), but the routine that calculates the screen zones simply skips past this icon.

```

Procedure _CHECKJOYSTICK
  If MARIO(2) <> 2

    Rem *** Has joystick been pushed right?
    If Joy(1) and 4
      If MARIO(2)=0
        MARIO(0)=MARIO(0)-SPEED
        DIRECTION=-1
        FRAME=FRAME or $8000

```



```
        End If
    End If

    Rem *** Has joystick been pushed left?
    If Joy(1) and 8
        If MARIO(2)=0
            MARIO(0)=MARIO(0)+SPEED
            DIRECTION=1
            FRAME=FRAME and %111
        End If
    End If

    Rem *** Stop animation if Mario is standing still
    If Joy(1)<3
        If MARIO(2)=0
            DIRECTION=0
            FRAME=FRAME and $8000
            FRAME=FRAME+3
        End If
    End If

    Rem *** Make Mario jump if fire button pressed
    If Joy(1) and 16
        If MARIO(2)=0
            MARIO(2)=1
            XOFFSET=MARIO(0)
            YOFFSET=MARIO(1)
        End If
    End If

    Rem *** Start Mario jumping...
    If MARIO(2)=1
        _JUMP
        FRAME=FRAME and $8000
        FRAME=FRAME+3
    End If
    If MARIO(0)>320
```

```

        MARIO(0)=320
    End If
    If MARIO(0)<0
        MARIO(0)=0
    End If
End If
End Proc

```

If you think that you've seen this routine before, you'd be right – the '\_CHECKJOYSTICK' procedure is virtually identical to the 'Jump Bob' procedure we covered in chapter 8. What it does is to read the position of the joystick and act accordingly. Most platform games don't use the 'up' and 'down' joystick positions and so these are ignored – the game simply traps the 'left', 'right' and 'fire' events and updates the Mario data structure accordingly. If the player presses the fire button, the sprite's status variable ("MARIO(2)") is set to one and the program jumps to a procedure called '\_JUMP' that makes it jump.

```

Procedure _JUMP
    HEIGHT=160 : Rem *** Maximum height of Jump
    WIDTH=80 : Rem *** Width of Jump
    JUMPSPEED=SPEED*2 : Rem *** Speed of Jump

```



```

Degree

```

```

Rem *** Is Mario still jumping?
If MARIO(3)<181

```

```

    Rem *** Jump left
    If DIRECTION=-1
        X=Cos(MARIO(3))*WIDTH/2
        MARIO(0)=XOFFSET+X-WIDTH/2
    End If

```

```

    Rem *** Jump Right
    If DIRECTION=1
        X=-Cos(MARIO(3))*WIDTH/2

```

```

        MARIO(0)=XOFFSET+X+WIDTH/2
    End If

    Y=-Sin(MARIO(3))*HEIGHT/2

    MARIO(1)=YOFFSET+Y
    MARIO(3)=MARIO(3)+JUMPSPEED
Else
    Rem *** Reset angle and status
    MARIO(1)=YOFFSET
    MARIO(2)=0
    MARIO(3)=0
End If
End Proc

```

The ‘\_CHECKJOYSTICK’ procedure isn’t the only routine that we have stolen from the ‘Jump Bob’ program in chapter 8 – the ‘\_JUMP’ procedure is based around that program too! It works by calculating the angle of the jump in a number of steps which are added to the player’s sprite’s ‘angle’ variable in its data structure. The procedure calculates the position of the sprite using the angle variable which is passed to the ‘Sin()’ and ‘Cos()’ functions. The angle variable starts with an initial value of 1 and as the jump is processed, it increases to a maximum of 180 by adding the ‘speed’ of the jump to the angle. If you decrease the speed of the jump, more steps will be calculated for the jump and the sprite will appear to move slower.

This routine can easily be tweaked to suit your own particular needs by changing the values of the ‘HEIGHT’, ‘WIDTH’ and ‘JUMPSPEED’ variables. Increasing the value of the ‘WIDTH’ variable, for example, will make the sprite jump further and increasing the value of the ‘HEIGHT’ variable will make it jump that bit higher.

```

Procedure _CHECKFALL

```

```

    FALLSPEED=5

```

```

    Rem *** Is Mario already jumping?

```



```

If MARIO(2)<>1

    Rem *** Is Mario standing on a platform?
    If Zone(MARIO(0),MARIO(1))=0

        Rem *** Make him drop down
        MARIO(1)=MARIO(1)+FALLSPEED
        MARIO(2)=2
    End If

    Rem *** Has Mario dropped off bottom of screen?
    If MARIO(1)>256
        Rem *** Wrap him around to top of screen
        MARIO(1)=0
    End If
End If
End Proc

```

The ‘\_CHECKFALL’ procedure is responsible for checking when the player’s sprite steps off a platform. This isn’t as involved as it sounds – all the routine does is to start by checking to make sure the sprite isn’t half way through a jump – if it is, then it’ll still have the momentum to carry on flying through mid-air. If it has walked off a platform or the jump has run its course and the sprite hasn’t landed on a platform, the routine then decreases its ‘Y’ position by a fixed amount held in the variable ‘FALLSPEED’. If the sprite continues to fall and it drops off the bottom of the screen, it wraps back around to the top in true ‘Bubble Bobble’ style. You could easily change this particular section of the code so that the sprite dies if you so wish.

Procedure \_CHECKLANDED

```

Rem *** Is Mario falling?
If MARIO(2)=2
    A=Zone(MARIO(0),MARIO(1))

    Rem *** Has he landed on a platform?

```



```
    If A<>0
        MARIO(1)=PLATFORM(A-1,1)+1
        MARIO(2)=0
    End If
End If

Rem *** Is Mario falling?
If MARIO(2)=1 and MARIO(3)>90
    A=Zone(MARIO(0),MARIO(1))

    Rem *** Has Mario landed?
    If A<>0
        MARIO(1)=PLATFORM(A-1,1)+1
        MARIO(2)=0
        MARIO(3)=0
    End If
End If
End Proc
```

The ‘\_CHECKLANDED’ procedure works in conjunction with the ‘\_CHECKFALL’ procedure to stop the sprite from falling when it lands on a platform. It’s very simple indeed. All it does is to check whether the sprite is already falling and if it is, the ‘Zone()’ function is used to check whether the object’s ‘hot spot’ is inside a screen zone. If it is, then the sprite has landed on a platform and its ‘Y’ position is modified to reflect this. Note how the position of the platform is read from the ‘PLATFORM()’ array that we set up at the start of the program – this is done just to ensure that if the sprite isn’t directly in line with the platform (it could be two or three pixels past the top of the platform), its position is correctly set so that it stands on the exact ‘Y’ position required for that platform.

The procedure is also used to check when the sprite has managed to successfully jump onto a new platform. It will only land on a platform, however, if half of the jump is complete (indicated by a value of 90). Because the jump effectively forms a path in the shape of a semi-circle, a value of 90 indicates the sprite has jumped as high as possible. This allows it to jump through one platform to another platform that is above



the first platform. Feel free to play around with this section of code to get the results that you want for your platform game.

```

Procedure _MOVEBADDIE
  For A=0 To NUMBADDIES
    If BADDIE(A,1)=0
      BADDIE(A,0)=PLATFORM(BADDIE(A,3),0)+8
      BADDIE(A,1)=PLATFORM(BADDIE(A,3),1)
    Else
      BADDIE(A,0)=BADDIE(A,0)+BADDIE(A,2)
      If BADDIE(A,0)>PLATFORM(BADDIE(A,3),2)-8
        BADDIE(A,2)=-BADDIE(A,2)
      End If
      If BADDIE(A,0)<PLATFORM(BADDIE(A,3),0)+8
        BADDIE(A,2)=-BADDIE(A,2)
      End If
    End If
  Next A
End Proc

```



The ‘\_MOVEBADDIES’ procedure is pretty intelligent too. Most platform games that I’ve seen written in AMOS previously insisted that you set the exact co-ordinates of the baddies on screen. My routine, however, simply needs to be fed the number of the platform that the baddie inhabits. Because the exact positions of the platforms has already been stored in the ‘PLATFORM()’ array, the routine simply calculates where the baddie should appear using the information held in this array.

The routine starts by checking to see whether the screen positions of the baddies have already been calculated. If they haven’t, then they’re calculated on the spot. If they have been calculated, however, the baddie is moved by adding its speed to its ‘X’ screen position. If the baddie reaches either end of a platform, its speed is reversed so that the next time it is moved, it’s ‘X’ position will be increased or decreased appropriately.

```
Procedure _CHECKCOLLISIONS
  STATUS=Bob Col(0,1 To NUMBADDIES)
  If STATUS=-1
    Rem *** Insert your collision code here!...
    Bell
  End If
End Proc
```



Finally, we have the ‘\_CHECKCOLLISIONS’ procedure, which checks to see if any collisions have occurred between the player’s sprite and any baddies that are wondering around on the platforms. This procedure is far from complete – as it is, it simply plays the AMOS ‘Bell’ sound if a collision takes place. I’ll leave it to you to extend the routine so that the player’s sprite is actually killed by any collisions with baddies. Good luck!



# Adventure games

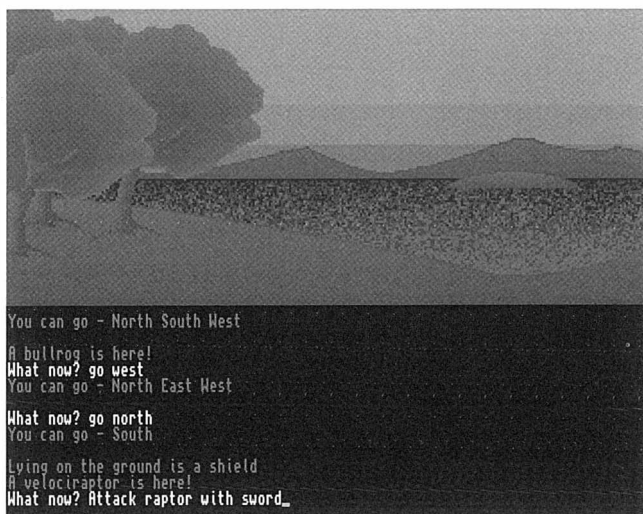
- Designing an adventure
- Writing a parser
- Moving around 'locations'
- Handling objects and monsters
- Adding graphics

**D**espite the genre's age, Adventure games continue to be popular amongst more thoughtful games players. Games like Magnetic Scrolls' 'The Pawn' and Infocom's infamous 'Hitch Hikers Guide to the Galaxy' combine the spirit of adventuring with good old puzzle-solving. Whereas your average arcade game can usually be completed within a matter of hours, adventure games can drag the player into a whole new world that can take literally months to explore.

It's no surprise, then, that adventure games are still a popular choice amongst would-be games programmers. Quite a few so-called 'Adventure Creators' have been released over the past couple of years or so (including Incentive's brilliant 'Graphic Adventure Creator'), but very few ever made it onto the Amiga (shame, I say!). The nearest Amiga users ever got to such programs was Aegis's rather disappointing 'Visionary', a programming language along the same lines as AMOS that was unfortunately never quite what its designers had hoped for, due mainly to the rather long-winded approach to adventure games programming that it employed.

Fret not, however, because good old AMOS can handle adventure games too! What's more, writing adventure games isn't quite as difficult as one might imagine. Not only can AMOS handle traditional text-based arcade

*If you've ever wanted to write your own adventure games, then why not use AMOS – it's simpler than you think!*



games like those produced by Infocom (pre-‘Shogun’, that is), but AMOS’s special qualities will even let you produce adventure games complete with colourful graphics, sound effects (imagine being able to actually hear the scene around you) and even animations!

---

## The parser

Writing adventure games is somewhat different from arcade games, as the time that the game takes to run is not so critical – whereas an arcade game must update the entire game within a 50th of a second, adventures games can take as long as they want (to a extent).

The heart of every adventure game is the ‘parser’, a routine that takes the text that the user enters and makes sense of it. I’m sure we’ve all played adventure games that pop a little ‘What now?’ prompt up onto the screen every time the game is expecting an instruction – well, it’s the ‘parser’ that makes sense of what you type. Your Amiga doesn’t understand human language, so writing an intelligent parser is an art form in itself.

All parsers work by assuming that each and every sentence you feed them has a very strict format. It’s a bit like a programming language interpreter really – unless you type in commands in the exact form that the interpreter is expecting, you’ll get a ‘Syntax Error’. Parsers impose exactly the same limitation – if, for example, you were to enter something strange like ‘Attack man knife with’, a human could easily understand what you were trying to say by sorting the words into their correct order, but computers still aren’t intelligent enough to do this sort of thing themselves. Unless you type the command exactly as the parser expects, you’ll get a short message informing you politely that the computer can’t make head nor tail of your entry.



If, on the other hand, you entered the command as it should be (‘Attack Man with Knife’), the parser will have a much better chance of understanding it. All parsers work by slicing the sentence that you feed them into a series of individual words that are placed into a dimensional array. If, for example, the parser were to work on our command ‘Attack man with knife’, the array would contain four words – ‘Attack’, ‘man’, ‘with’, ‘knife’. Once the parser has done this, it then compares the first

word with a long list of commands that the programmer has taught it. Most parsers contain at least twenty or so commands, but you could easily get by with a fairly minimal selection – ‘Go’, ‘Attack’, ‘Examine’, ‘Get’ and ‘Drop’ etc.

If the parser manages to find a match, it then jumps to a routine specifically designed to handle that particular command. What’s more, because the parser has interpreted the first word, it knows exactly what it should do with any extra words that follow it – in the above example, the parser would know that the second word contains the name of the person that is to be attacked and the fourth word contains the name of the object that it should use for the attack. It would therefore then compare the second word against a second dimensional array containing a list of all the various objects, monsters and people depicted in the game.

Obviously it’s no good just recognising the object that is to be attacked – the parser also needs to know if the person that the player has requested it to attack is actually nearby. Each attackable object, person or monster therefore has a number of extra items of information attached to it – its location within the game (is it in the same room?), how hard it is to kill and perhaps a number of extra items of information that will tell the parser how the object, person or monster will react to being attacked (will it run, stand and fight or just take it lying down?).

---

## Objects and monsters

Just like an arcade game, every object and person in your arcade game has a data structure assigned to it that holds all the information required to handle it. If you were writing a fantasy adventure along the lines of ‘Dungeons & Dragons’, you might have a skeletal warrior that was controlled by a data structure that contained the following information.

<b>Name</b>	-	<b>Skeletal Warrior</b>
<b>Location</b>	-	<b>Room 16</b>
<b>Hit Points</b>	-	<b>10 (Between 1 and 20)</b>
<b>Strength</b>	-	<b>5 (Between 1 and 10)</b>
<b>Skill</b>	-	<b>4 (Between 1 and 10)</b>
<b>Alignment</b>	-	<b>Hostile</b>

If you're feeling particularly adventurous (pardon the pun), you could easily make the creatures within your adventure game move around the game map simply by updating their position each time the player makes a move. Most modern adventures allow creatures to wander, so why not try it for yourself! Anyway, where were we...?

If the parser managed to find a match with the name of the person that the player wants to attack, it then checks whether the person is in the same room. If they aren't, then there's little point trying to attack them! Once the parser has managed to recognise the person that is being attacked, it then checks the forth word to see what object the player wishes to attack the person with. Once again, an array is checked that contains a list of all the objects in the game. If the player doesn't have the object that they've requested to use, a message is once again displayed telling the player that the action that they've requested is not possible ('You don't have the knife!', for example).

If everything went to plan, the parser will have accepted your command. All that now remains is to carry out whatever action is required to fulfil that command. In the case of our example, the parser would then call a routine that handled combat between the player and the person that they've decided to pick upon. If the player survived the attempted attack (never attack a Bullrog with a garden hose!), the parser would go back to the beginning and start again by displaying the 'What now?' prompt.

---

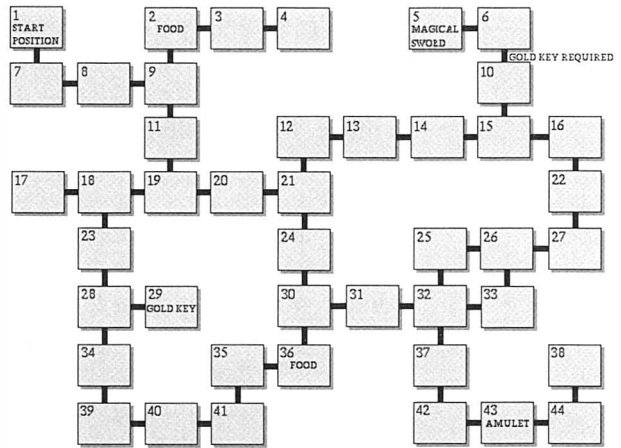
## Moving around locations

One of the most important aspects of any adventure game is the ability to move around the various 'rooms' that form the adventure's virtual world. This is perhaps the most important aspect of any adventure – after all, if the player's computerised alter ego stands in the same place all the way through, it's hardly going to be a very exciting game! Even on an unexpanded 512K A500 (remember them?), it's possible to create some quite substantial game 'worlds' that the player can wander around and explore at their leisure.

Before you sit down and code your latest Infocom-killer, however, it's well worth taking time to design your game world. This is very important



Before you can write an adventure game, you need to start by designing the game map on paper (or, as we've done here, in Deluxe Paint).



indeed, as you need some form of reference material to feed in all the information that the movement routine will use. The size of the game map is entirely up to you, but bear in mind that each and every location needs its own textual description, which must be fed in by hand, along with all the various movement attributes that will be needed.

So what do you need to get started? First of all you'll need to draw up a game map similar to the one shown in this chapter. Every single square within the map is a location that the player can move to via one or more of the six directions of movement – north, south, east, west, up and down. To make life somewhat more complicated, you can also control the player's movement by making certain paths (a path is an imaginary line that joins two locations) either one-way or 'restricted entry', in which case the player needs a certain object (a gold key to unlock the door, for example) to travel along that particular path. The paths themselves will be completely transparent to the player – they're simply there to serve as a reference when keying in all the location data.

Once you've drawn up your game map, you need to key all the map data into the computer. You'll need to create a dimensional array that holds all the map's path data using a line such as 'Dim ROOMS(100,6)'. This would create a dimensional array capable of handling 100 different locations, each of which has 6 different items of information attached to it. How many locations you use in your game is entirely up to you – if

you're feeling adventurous and enjoy a little bit of hard work, there's absolutely no reason why you couldn't create an adventure game with literally thousands of different locations! Each location has attached to it 6 different items which contain the movement attribute for each direction of movement – one for north, one for south etc.

Each of these movement attributes act as a sort of pointer that tells the adventure game parser which location that player would move to if they opted to move in a particular direction. Say, for example, the player is currently in location 10 and a movement to the north would take them to location 12. The movement attribute that holds the north direction for location 10 would therefore contain a value of 12. Similarly, the opposite would be true – the attribute that holds the south direction for location 12 would contain a value of 10, allowing the player to move south and then move north again if they so wish. It's important to remember which movement attribute holds the information for a particular direction – reading the value for the north attribute when the player wants to move south could be disastrous (who knows what lurks in that direction?).

It's also possible to add graphics to your adventures so that the player can not only read a description of each location, but also see it! AMOS makes this sort of thing very easy indeed thanks to its ability to open multiple screens. All you need to do is to open two separate screens – one for your graphics and one for your text – and then arrange them so that the text screen appears below the graphics screen. You can then extend your game map array so that it not only contains all the various movement attributes for each location, but also the filename of a picture held on disk that is to be displayed for that particular location.



**Adding  
graphics**

Getting even more adventurous still, why not create a bank of bobs for all the objects and lifeforms that inhabit your adventure? By simply checking what objects and lifeforms are in a particular location each time the player moves to that location, you could paste the bob image of that particular object or lifeform onto the graphic screen so that the player not only sees the location that they've moved to, but any objects or creatures that they encounter too! Who ever said that AMOS couldn't be used to write decent adventures?! Humbug to the lot of 'em...

Phew! What a lot of theory. Anyway, here's a listing that should provide you with the building blocks of a very flexible adventure game parser. Let's brake the code down into more manageable chunks and take a look at each in turn.

```

Rem *** Simple Adventure Parser Demonstration
Rem *** Filename - AdventureParser.AMOS

Rem *** Open picture screen...
Screen Open 0,320,150,32,Lowres
Flash Off : Curs Off

Rem *** Open text screen...
Screen Open 1,640,100,4,Hires
Cls 0 : Paper 0 : Pen 1
Screen Display 1,128,196,,
Palette $0,$FFF,$F00,$F

Rem *** Number of commands, objects and lifeforms
NOUNS=6 : OBJECTS=3 : LIFEFORMS=2 : NUMLOCATIONS=6

Rem *** Initialise commands array
Dim NOUN$(NOUNS)

Rem *** Initialise game map array
Rem *** MAP$(n,0) = North direction
Rem *** MAP$(n,1) = South direction
Rem *** MAP$(n,2) = East direction
Rem *** MAP$(n,3) = West direction
Rem *** MAP$(n,4) = Picture filename
Dim MAP$(NUMLOCATIONS,5)

Rem *** Initialise objects array
Rem *** OBJECT$(n,0) = Object name
Rem *** OBJECT$(n,1) = Location of object
Rem *** OBJECT$(n,2) = Status (0 = On floor 1 = Picked Up)
Dim OBJECT$(OBJECTS,3)

```



```
Rem *** Initialise life (monsters etc) array
Rem *** LIFE$(n,0) = name of lifeform
Rem *** LIFE$(n,1) = Location of lifeform
Rem *** LIFE$(n,2) = Strength of lifeform (0 = Dead)
Dim LIFE$(LIFEFORMS,3)

Rem *** Initialise words array
Dim WORDS$(5) : WORDCOUNT=0

Global NOUNS,OBJECTS,LIFEFORMS
Global NOUN$(),OBJECT$(),LIFE$(),WORDS$()
Global WORDCOUNT,ROOM,MAP$(),NUMLOCATIONS

__INITMAP
__INITPARSER

ROOM=1
__ROOMINFO[ROOM]

Do
  Input "What now? ";COMMAND$
  If COMMAND$<>" "
    __PARSE[COMMAND$]
  End If
Loop
```

Before the main bulk of the adventure game code starts, a fair amount of setting up is necessary. Once the two screens have been opened (screen 0 holds the picture of the location and screen 1 holds the adventure text), four variables are defined that tell the game how many words the parser can handle ('NOUNS'), how many objects are scattered around the adventure map ('OBJECTS'), how many lifeforms there are ('LIFEFORMS') and how many locations are in the game map ('NUMLOCATIONS'). These variables are then used to set up all the various data structures that are required to handle the game. These variables and data structures are then made global before passing control to two procedures ('\_\_INITMAP' and '\_\_INITPARSER') that set the game

up. The text for the first location is then displayed by calling the ‘\_ROOMINFO’ procedure.

Once all this setting up is complete, the game then starts by entering a very small loop that asks the user to enter a command and then passes it to the ‘\_PARSE’ procedure, which interprets it.

#### Procedure \_INITMAP



```

Rem *** Read direction data into map array

Restore MAPDATA
For A=0 To NUMLOCATIONS-1
    Read MAP$(A,0) : Rem *** North
    Read MAP$(A,1) : Rem *** South
    Read MAP$(A,2) : Rem *** East
    Read MAP$(A,3) : Rem *** West
    Read MAP$(A,4) : Rem *** Picture filename
Next A

MAPDATA:
Data "0","3","0","0","AMOSBOOK:Pictures/ADV.Room1"
Data "0","4","0","0","AMOSBOOK:Pictures/ADV.Room2"
Data "1","0","4","6","AMOSBOOK:Pictures/ADV.Room3"
Data "2","5","0","3","AMOSBOOK:Pictures/ADV.Room4"
Data "4","0","0","0","AMOSBOOK:Pictures/ADV.Room5"
Data "0","0","3","0","AMOSBOOK:Pictures/ADV.Room6"
End Proc

```

The ‘\_INITMAP’ procedure is very important indeed. What it does is to configure the array that holds the game map data including such details as the paths between locations and the filenames of the picture files associated with each location. A simple loop is used to read this information which is held inside the program using ‘Data’ statements.



```
Procedure _INITPARSER
  Restore NOUNS
  For A=0 To NOUNS-1
    Read NOUN$(A)
  Next A

  Restore OBJECTS
  For A=0 To OBJECTS-1
    Read OBJECT$(A,0)
    Read OBJECT$(A,1)
    Read OBJECT$(A,2)
  Next A

  Restore LIFE
  For A=0 To LIFEFORMS-1
    Read LIFE$(A,0)
    Read LIFE$(A,1)
    Read LIFE$(A,2)
  Next A

  NOUNS:
  Data "GO","ATTACK","INVENTORY"

  OBJECTS:
  Data "BOOK","2","0"
  Data "SHIELD","1","0"
  Data "SWORD","1","1"

  LIFE:
  Data "VELOCIRAPTOR","1","1"
  Data "BULLROG","4","20"
End Proc
```

The ‘\_INITPARSER’ procedure sets up all the data structures that are needed to handle all the nouns, objects and lifeforms that the game uses. For a more in-depth look at the structure of each data structure, refer to the start of the program.



```

Procedure _PARSE[COMMAND$]

    Rem *** Slice command string into words

    WORDCOUNT=0
    COMMAND$=Upper$(COMMAND$)
    WORD$=""

    For A=1 To Len(COMMAND$)
        LETTER$=Mid$(COMMAND$,A,1)

        If LETTER$<>" "
            WORD$=WORD$+LETTER$
        End If

        If LETTER$=" " and WORDCOUNT<5 and LASTLETTER$<>" "
            WORDS$(WORDCOUNT)=WORD$
            WORDCOUNT=WORDCOUNT+1
            WORD$=""
        End If

        If A=Len(COMMAND$) and WORDCOUNT<5
            If WORD$<>""
                WORDS$(WORDCOUNT)=WORD$
            End If
        End If

        LASTLETTER$=LETTER$
    Next A

    Rem *** Compare first word with list of commands

    NOUNNUM=-1
    For A=0 To NOUNS-1
        If WORDS$(0)=NOUN$(A)
            NOUNNUM=A
        End If
    
```

```
Next A

Rem *** Was a match found?

If NOUNNUM>-1
  If NOUNNUM=0
    _GO
  End If
  If NOUNNUM=1
    _ATTACK
  End If
  If NOUNNUM=2
    _INVENTORY
  End If

  Rem *** Rest of parser command handling code...
Else
  Pen 2
  Print "**** Sorry, I don't understand!"
  Print
  Pen 1
End If
End Proc
```

The ‘\_PARSE’ procedure is responsible for slicing up the command that the player enters into a series of individual words that are stored in the global array ‘WORDS\$()’. Each word is extracted by stepping through the entire command a character at a time, appending each letter to a temporary variable until a space is found. If a space is found, then the parser assumes that a whole word has been extracted and it’s placed into the ‘WORDS\$()’ array. Once this is done, the procedure then compares the first word in the array against the list of nouns it has stored in the ‘NOUN\$()’ array. If a match is found, then the first word is accepted and the routine branches off to the procedure that is designed to handle that particular noun. If the user were to enter ‘Attack’, for example, the routine would branch to the ‘\_ATTACK’ procedure which is solely responsible for handling the rest of the user’s command.





```

Procedure _ROOMINFO[ROOM]
  Rem *** Load and display picture if there is one...
  If MAP$(ROOM-1,4) <> ""
    Screen 0
    Load Iff MAP$(ROOM-1,4)
    Screen 1
  End If

  Rem *** Load and display location text here...

  Rem *** Display available directions...
  Pen 2
  Print "You can go - ";
  For A=0 To 3
    Read DIRECTION$
    If Val(MAP$(ROOM-1,A)) <> 0
      Print DIRECTION$;
    End If
  Next A
  Pen 1
  Print : Print

  Rem *** Display objects in room...
  Pen 2
  For A=0 To OBJECTS-1
    If Val(OBJECT$(A,1))=ROOM and Val(OBJECT$(A,2)) <> 1
      Print "Lying on the ground is a ";
      Print Lower$(OBJECT$(A,0))
    End If
  Next A
  Pen 1

  Rem *** Display lifeforms in room...
  Pen 2
  For A=0 To LIFEFORMS-1
    If Val(LIFE$(A,1))=ROOM

```

```

                Print "A ";Lower$(LIFE$(A,0));" is here!"
            End If
        Next A
    Pen 1

    Data "North ","South ","East ","West"
End Proc

```

The ‘\_ROOMINFO’ procedure is responsible for displaying all the information that the player needs to know about a location once they move to it. The procedure starts by checking to see whether a picture is attached to the current location by checking the ‘MAP\$()’ array. If a filename is found, the picture is loaded into screen 0. I’ve left out the next section of code that should display the description that is associated with the location. You can easily add this yourself by either reading in a section of text from disk or by displaying text held in an array specifically designed to hold the location text.

Most adventure games only tell you what directions you can move in and what objects and lifeforms are in the location if you specifically ask them to, but our game uses a more direct (and certainly more helpful) approach. All the possible moves, objects and lifeforms are displayed by scanning through the appropriate arrays. In the case of objects and lifeforms, the ‘location’ variable that is associated to each is checked to see whether it matches the current location. A further check is needed for the objects – because each object has a status variable associated with it that tells the game if the object is either lying on the floor or whether the player has picked up the object, the routine makes sure that only those objects that haven’t been picked up are listed.

```

Procedure _GO
    MOVEDFLAG=0

    For A=0 To 3
        Read DIRECTION$

        Rem *** Has direction been understood?

```



```

If Upper$(WORDS$(1))=DIRECTION$

    Rem *** Is it possible to move in that direction?
    If Val(MAP$(ROOM-1,A))<>0
        ROOM=Val(MAP$(ROOM-1,A))
        _ROOMINFO[ROOM]
        MOVEDFLAG=1
    Else
        Pen 2
        Print "*** Sorry, you can't move ";
        Print "in that direction!"
        MOVEDFLAG=1
        Pen 1
    End If
End If
Next A
If MOVEDFLAG=0
    Pen 2
    Print "*** Go in which direction?"
    Pen 1
End If

Data "NORTH","SOUTH","EAST","WEST"
End Proc

```

The first of the specific routines to handle the commands that the user can enter is the ‘\_GO’ procedure that handles the noun of the same name. The procedure starts by attempting to understand the direction that the player has requested. If the direction is understood, the routine then checks to see if the player can actually move in that direction. If the move is possible, then the player’s command is accepted and the player’s current location is changed.

If, on the other hand, either the direction is not understood or the player cannot move in that direction, an error message is displayed.



## Procedure \_ATTACK

```

Rem *** Compare second word with list of lifeforms
LIFENUM=-1
For A=0 To LIFEFORMS-1
  If WORDS$(1)=LIFE$(A,0)
    LIFENUM=A
  End If
Next A

Rem *** Has lifeform been identified?

If LIFENUM>-1

  Rem *** Is the lifeform in the same room?
  If Val(LIFE$(LIFENUM,1))=ROOM

    Rem *** Compare fourth word with list of objects
    OBJNUM=-1
    For A=0 To OBJECTS-1
      If WORDS$(3)=OBJECT$(A,0)

        Rem *** Does player have that object?
        If OBJECT$(A,2)="1"
          OBJNUM=A
        Else
          OBJNUM=-2
        End If
      End If
    Next A

    Rem *** Was the object found?
    If OBJNUM>-1
      Pen 2
      Print "**** You attack the ";Lower$(WORDS$(1));
      Print " with your ";Lower$(WORDS$(3))
      Print
    End If
  End If
End If

```

```

        Pen 1

        Rem *** Jump to combat routine
        Rem *** I'll leave this for you to write!
    Else
        If OBJNUM=-1
            Pen 2
            Print "**** Attack ";Lower$(WORDS$(1));
            Print " with what?"
            Print
            Pen 1
        End If
        If OBJNUM=-2
            Pen 2
            Print "**** You don't have the ";
            Print Lower$(WORDS$(3));"!"
            Print
            Pen 1
        End If
    End If
Else
    Pen 2
    Print "**** The ";Lower$(WORDS$(1));
    Print " isn't here!"
    Print
    Pen 1
End If
Else
    Pen 2
    Print "**** Attack what?"
    Print
    Pen 1
End If
End Proc

```

The ‘\_ATTACK’ procedure is responsible for handling the command of the same name. Although not complete, it does interpret the entire command string and it would be a fairly minor task to write a procedure

that handles combat between the player and the lifeform that they choose to attack.

The routine starts by trying to identify the lifeform that the player has attempted to attack by comparing the second word in the 'WORDS\$()' array against the list of lifeforms held in the 'LIFEFORM\$()' array. If a match is found, the procedure then checks to make sure that the lifeform is in the same location as the player. If all of this checking proves positive, the procedure then compares the fourth word against the list of objects in the 'OBJECT\$()' array to see if the object that the player has selected to attack the lifeform with exists. Even if a match is found, the procedure must check to make sure that the player is holding that object.

If the object is found and the lifeform is in the same room, the combat routine would then be called. If, during all these checks, something does not prove to be true, an appropriate error message is displayed.

```
Procedure _INVENTORY
  Pen 2
  Print "You are carrying - ";

  EMPTY=-1

  For A=0 To OBJECTS-1
    If OBJECT$(A,2)="1"
      Print OBJECT$(A,0);" ";
      EMPTY=1
    End If
  Next A

  If EMPTY=-1
    Print "Nothing"
  End If

  Print
  Pen 1
End Proc
```



The last procedure is the ‘\_INVENTORY’ procedure that handles the commands of the same name. Unlike most adventure games, objects aren’t held in a temporary ‘inventory’ array – simply by changing an object’s status variable to a value of one, the program automatically assumes that the player is holding that object. The inventory is displayed using a loop that checks through every object in the game – if an object’s status is set to one, then the name of the object is displayed, otherwise it is ignored.

Obviously our demo adventure is very limited indeed – after all, it uses only six locations and a very limited number of commands, objects and lifeforms. Adding extra commands would be very easy indeed – the ‘take’ and ‘drop’ commands, for example, could be implemented simply by changing the status variable of an object from zero to one when the player wants to pick up an object and from one to zero when the player wants to drop an object.

# Appendix A: Useful routines

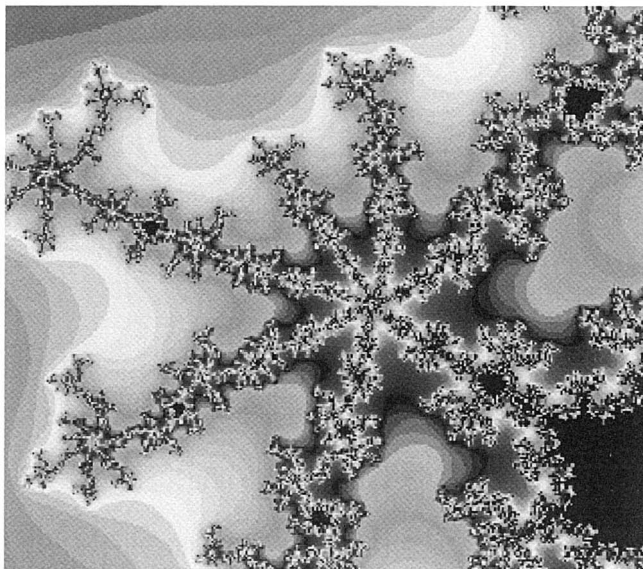
- Mandelbrot generator
- 'Splerge' effect
- Parallax starfields
- Multitasking text input
- Co-ordinates finder
- High Score routine
- Bubble sorting arrays



## Mandelbrot Generator

Although not directly relevant to games programming, fractal graphics are still a fascinating area of computing. They give you the ability to generate amazing pictures of virtually unlimited complexity with absolutely no artistic skills whatsoever! The following program allows you to create Mandelbrot images and even zoom in for greater detail.

*Create stunning fractal images like this using the Mandelbrot generator program below.*



```
Rem *** AMOS Mandelbrot Set Explorer
Rem *** By Jason Holborn
Rem *** Based on an AmigaBASIC routine by Conrad Bessant
Rem *** Filename - Mandelbrot.AMOS
```



```
SCRX=320
SCRY=256
```

```
Screen Open 0,SCRX,SCRY,32,Lowres
Flash Off : Curs Off : Cls 0
```

```
Rem *** Load default picture...
```

```
Load Iff "AMOSBOOK:Pictures/Mandelbrot.IFF"

Rem *** Set initial boundaries for Mandelbrot
XMIN#=-2
XMAX#=0.8
YMIN#=-1.2
YMAX#=1.2
MKITERS=96 : Rem *** Level of detail

Do
  Rem *** Mandelbrot 'Zoom' facility....
  ZOOM=0 : ERR=0
  Repeat
    If Mouse Key=1
      X1=X Screen(X Mouse)
      Y1=Y Screen(Y Mouse)
      Repeat
        X2=X Screen(X Mouse)
        Y2=Y Screen(Y Mouse)
        If X2>X1 and Y2>Y1
          Gr Writing 3
          Ink 15
          Box X1,Y1 To X2,Y2
          Wait Vbl
          Box X1,Y1 To X2,Y2
          ERR=0
        Else
          ERR=1
        End If
      Until Mouse Key=0
    If ERR=0
      Gr Writing 0
      Box X1,Y1 To X2,Y2

      XRANGE#=(XMAX#-XMIN#)/SCRX
      YRANGE#=(YMAX#-YMIN#)/SCRY
```

```
        XMAX#=(XRANGE#*X2)+XMIN#
        XMIN#=(XRANGE#*X1)+XMIN#
        YMAX#=(YRANGE#*Y1)+YMIN#
        YMIN#=(YRANGE#*Y2)+YMIN#

        ZOOOM=1
    End If
End If

Rem *** Has user pressed 'S' key to save picture?
If Key State(33)=-1
    FILENAME$=Fsel$("RAM:**", "", "Save Picture", "")
    If FILENAME$<>""
        Save Iff FILENAME$
    End If
End If

Rem *** Has user pressed 'G' key to generate image?
If Key State(36)=-1
    ZOOOM=1
End If

Rem *** Has user pressed 'R' key to reset mandelbrot?
If Key State(19)=-1
    XMIN#=-2
    XMAX#=0.8
    YMIN#=-1.2
    YMAX#=1.2

    Rem *** Load default picture...
    Load Iff "AMOSBOOK:Pictures/Mandelbrot.IFF"
End If

Until ZOOOM=1 and ERR=0

    _MANDELBROT[XMIN#,XMAX#,YMIN#,YMAX#,MXITERS,SCRX,SCRY]
Loop
```

```
Procedure _MANDELBROT [XMIN#, XMAX#, YMIN#, YMAX#, MXITERS, SCRX, SCRY]
  X=0
  Y=SCRY-1

  XRANGE#=(XMAX#-XMIN#)/SCRX
  YRANGE#=(YMAX#-YMIN#)/SCRY

  Rem *** Generate Mandelbrot
  For A#=XMIN# To XMAX# Step XRANGE#
    For B#=YMIN# To YMAX# Step YRANGE#

      Rem *** Set initial values of variables
      P#=0
      Q#=0
      ITERATION=0

      While P#*P#+Q#*Q#<4 and ITERATION<MXITERS
        PNEW#=P#*P#-Q#*Q#+A#
        QNEW#=2*P#*Q#+B#
        P#=PNEW#
        Q#=QNEW#
        ITERATION=ITERATION+1

        If Key State(64)
          Pop Proc
        End If
      Wend

      Plot X,Y,ITERATION
      Y=Y-1
    Next B#
    Y=SCRY-1
    X=X+1
  Next A#
End Proc
```

## 'Splerge' effect

Adding weird and wonderful screen transition effects to your games can add that little bit of extra sparkle that sets a great game aside from an average game. One of the most popular (and most impressive!) is the 'Splerge' effect which draws up a picture almost as if it were 'melting' onto the screen. Prepare to be amazed!

*One of the most popular screen wipes amongst professional programmers is the 'splerge' effect.*



```
Rem *** Fast Splerge Routine
Rem *** By Jason D Banks
Rem *** Updated by Jason Holborn
Rem *** Filename - Splerge.AMOS
```



```
Screen Open 1,320,256,32,Lowres
Flash Off : Curs Off : Cls 0
Load Iff "AMOSBOOK:Pictures/DemoPicture.IFF"
```

```
Screen Open 2,320,256,32,Lowres
Curs Off : Flash Off : Cls 0
Get Palette 1
```

```
SPLERGE [0,1,2]
```

```
A$="" : Rem *** Type a key to start a splerge effect.
```

```
While A$<>"Q"
```

```
    A$=Upper$(Inkey$)
```

```
    If A$="1" Then SPLERGE[0,1,2]
```

```
    If A$="2" Then SPLERGE[1,1,2]
```

```
    If A$="3" Then SPLERGE[2,1,2]
```

```
    If A$="4" Then SPLERGE[3,1,2]
```

```
    If A$="5" Then SPLERGE[4,1,2]
```

```
    If A$="9" Then SPLERGE[8,1,2]
```

```
Wend
```

```
End
```

```
,
```

```
Rem *** SPLERGE Procedure
```

```
Rem *** SPEED = Speed of redraw (0=fastest)
```

```
Rem *** SOURCE = Source screen
```

```
Rem *** DEST = Destination screen
```

```
Procedure SPLERGE[SPEED,SOURCE,DEST]
```

```
    Screen SOURCE
```

```
    SOURCE_SIZE=Screen Height
```

```
    Screen DEST
```

```
    DEST_SIZE=Screen Height
```

```
    V=Min(SOURCE_SIZE,DEST_SIZE)
```

```
    Screen SOURCE
```

```
    SOURCE_SIZE=Screen Width
```

```
    Screen DEST
```

```
    DEST_SIZE=Screen Width
```

```
    H=Min(SOURCE_SIZE,DEST_SIZE)
```

```
    For LOP=V-1 To 1 Step -1
```

```
    _FILL[DEST,0,0,H,SOURCE,LOP]
    If SPEED
        Wait SPEED
    End If
Next LOP
End Proc

Procedure _FILL[DEST,SX,SY,WIDTH_X, SRC, LINE]
    Screen Copy SRC,0,LINE,WIDTH_X,LINE+1 To DEST,SX,SY

    STP=1 : COUNT=1

    While COUNT<LINE
        Screen Copy DEST,0,0,WIDTH_X,STP To DEST,0,STP
        STP=STP*2
        COUNT=COUNT+STP
    Wend

    If COUNT>LINE
        Screen Copy DEST,0,0,WIDTH_X,LINE-STP To DEST,0,STP
    End If

    If COUNT=LINE
        Screen Copy DEST,0,0,WIDTH_X,1 To DEST,0,LINE-1
    End If

    Wait Vbl
End Proc
```

## X-Plane Starfield

If you're writing a shoot 'em up based in space, then you can add that extra feeling of depth to your game by adding a 3D parallax starfield. Most routines of this type draw the stars using AMOS's 'Plot' command but my routine uses blitter objects instead. The advantage of this approach is that the stars do not interfere with the background graphics, so you're free to draw graphics onto the screen and load IFF backdrops without the starfield interfering with them. A word of warning, however - because this routine draws 40 blitter objects onto the screen in a single frame, you really really do need to compile it for smooth movement.

You can customise the routine quite a bit yourself. Simply change the values of the 'PLANES' and 'STARS' variables to alter the 'depth' of the starfield and the number of stars respectively. The starfield itself is completely random, so you'll never see the same starfield twice!

```
Rem *** Parallax X-Plane Starfield
Rem *** Compile this program for maximum speed
Rem *** Filename - X-Starfield.AMOS
```



```
PLANES=5 : Rem *** Levels of Parallax
STARS=40 : Rem *** Number of Stars
```

```
Dim X(STARS),Y(STARS),Z(STARS)
Global PLANES,STARS,X(),Y(),Z()
```

```
Screen Open 0,320,256,16,Lowres
Curs Off : Flash Off : Cls 0
```

```
Pen 2 : Paper 0
Locate 0,5 : Centre "Parallax Starfield"
```

```
Double Buffer : Autoback 0
Bob Update Off
```

```
Load "SOURCE:Star.ABK"
```



Get Sprite Palette

\_SETUPSTARS

Repeat

Bob Clear  
\_MOVESTARS

Bob Draw  
Screen Swap 0  
Wait Vbl

Until Mouse Key=1

End

Procedure \_SETUPSTARS

Rem \*\*\* Assign random speed and vertical  
Rem \*\*\* position to all stars  
For N=1 To STARS  
X(N)=320  
Y(N)=Rnd(256)  
Z(N)=Rnd(PLANES)+1

Next N

End Proc

Procedure \_MOVESTARS

For N=1 To STARS

Rem \*\*\* Update position of star  
Add X(N), -Z(N)

Rem \*\*\* Check that star hasn't left screen  
If X(N)<0

Rem \*\*\* If it has, generate new star

X(N)=320  
Y(N)=Rnd(256)  
Z(N)=Rnd(PLANES)+1

End If

```

    Rem *** Plot new star
    Ink 1
    Bob N,X(N),Y(N),1
Next N
End Proc

```

---

## Z-Plane Starfield

We've already covered the 'X' plane starfield, so all that remains is to take a look at a 'Z' plane starfield. If you've ever played a game like 'Elite', then you'll be instantly familiar with the 'Z' plane starfield. The effect gives the impression of flying through a starfield at high speed as viewed from the front of a spaceship. Anyway, here it is in all its AMOS glory.

```

Rem *** Z-field Starfield Effect
Rem *** Filename - Z-Starfield.AMOS

XRANGE=10 : Rem *** Max X axis speed
YRANGE=10 : Rem *** Max Y axis speed
NUM=50 : Rem *** Number of stars

Dim X(NUM),Y(NUM),XS(NUM),YS(NUM)
Global XRANGE,YRANGE,NUM,X(),Y(),XS(),YS()

Screen Open 0,320,256,2,Lowres
Flash Off : Cls 0 : Curs Off
Double Buffer : Autoback 1
Bob Update Off

Load "AMOSBOOK:ABKFiles/Star.ABK"
Get Sprite Palette

_SETUPSTARS

Repeat

```



```
Bob Clear
```

```
  _MOVESTARS
```

```
Bob Draw
```

```
Screen Swap 0
```

```
Wait Vbl
```

```
Until Mouse Key=1
```

```
Procedure _SETUPSTARS
```

```
  For A=0 To NUM
```

```
    X(A)=160 : Y(A)=128
```

```
    Randomize Timer
```

```
    XS(A)=(Rnd(XRANGE)+1)-(XRANGE/2)
```

```
    YS(A)=(Rnd(YRANGE)+1)-(YRANGE/2)
```

```
  Next A
```

```
End Proc
```

```
Procedure _MOVESTARS
```

```
  For A=0 To NUM
```

```
    Add X(A),XS(A)
```

```
    Add Y(A),YS(A)
```

```
  If X(A)<0 or X(A)>320 or Y(A)<0 or Y(A)>256
```

```
    X(A)=160 : Y(A)=128
```

```
    Randomize Timer
```

```
    XS(A)=(Rnd(XRANGE)+1)-(XRANGE/2)
```

```
    YS(A)=(Rnd(YRANGE)+1)-(YRANGE/2)
```

```
  End If
```

```
  Bob A,X(A),Y(A),1
```

```
Next A
```

```
End Proc
```

## Multitasking text input

Here's a handy routine if you've ever wanted to get the user to enter a string of text whilst AMOS gets on with other more important tasks – like animating sprites, drawing graphics and playing music, etc. Unlike AMOS's own 'Input' command, the following routine does not halt the execution of your program until the player presses the 'RETURN' key, so you could even (in theory at least!) have a game playing whilst the user enters text! Clever stuff, eh?

```

Rem ** GetInput Procedure
Rem ** Filename - GetInput.AMOS

_GETINPUT[1,3,10,-1]
ANSWER$=Param$

Locate 1,5
Print "The answer you typed was";Val(ANSWER$)
End

Rem ** _GETINPUT Procedure
Rem ** X= X coord of number to be entered
Rem ** Y= Y coord of number to be entered
Rem ** TIME = Max number of seconds allowed
Rem ** LIMIT = Maximum number of characters

Procedure _GETINPUT[X,Y,TIME,LIMIT]
  Curs Off

  TIME=(TIME+1)*50
  Timer=0

  If LIMIT=-1
    LIMIT=200
  End If

```



```
Repeat
  A$=Inkey$
  A=Scancode

  If A<>0
    If A=65
      STRLEN=Len(STRG$)
      If STRLEN>0
        STRG$=Left$(STRG$,STRLEN-1)
      End If
    Else
      If A<>0
        If Len(STRG$)<LIMIT
          STRG$=STRG$+A$
        End If
      End If
    End If
  End If

  Wait Vbl
  Locate X,Y : Print "Enter a number:"
  Locate X+17,Y : Print STRG$+"_ "

  Rem ** Put your code in here if you
  Rem ** would like AMOS to do something else

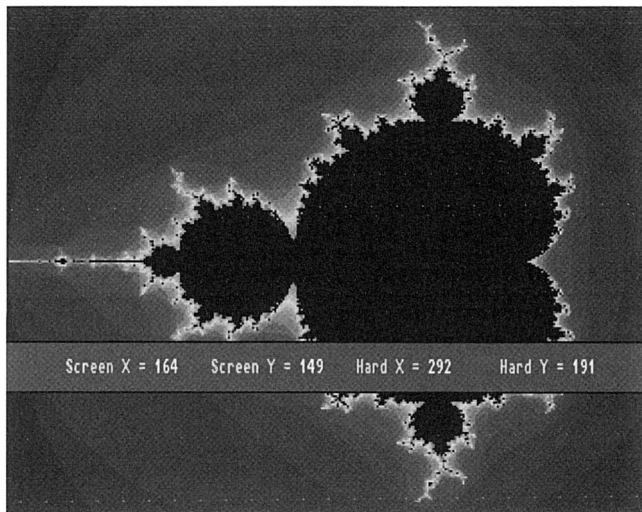
Until A=68 or A=67
End Proc[STRG$]
```

## Co-ordinate finder

Setting up screen displays can be a pain unless you're the sort of masochist that enjoys calculating the positions of bobs and sprites by hand. The following routine installs as a handy little programming tool that instantly comes on line whenever you press the 'F1' key on your Amiga keyboard. Just insert it into your game's main loop, press F1 and your program will be frozen. A tiny screen will then appear complete with read outs of the mouse pointer's current 'X' and 'Y' co-ordinates in both screen and hardware format. You can then calculate screen co-ordinates simply by moving the mouse pointer to the screen position you're interested in.

If you find that the co-ordinates screen obscures the area of the screen that you're interested in, then just use the 'Up' and 'Down' arrow keys on your Amiga keyboard to smoothly scroll the screen out of the way. Once you've finished, just press the 'F2' key and your program will continue running!

*Setting up screen zones and plotting bobs is considerably easier with this handy 'Co-ordinate Finder' utility.*





```
Rem *** Co-ordinate Finder
Rem *** Filename - CoordFinder.AMOS
Rem *** Do not use screen 7 in your own programs!
```

```
Do
```

```
Rem *** Add this line to your code...
  _FINDCOORDS[0]
```

```
Wait Vbl
```

```
Loop
```

```
Procedure _FINDCOORDS[SCRNO]
```

```
Rem *** Press F1 to start utility...
If Key State(80)=-1
```

```
  SCRYPOS=250
```

```
  Amal Freeze
  Limit Mouse
  Change Mouse 2
  Show
```

```
Rem *** Open up co-ords screen...
Screen Open 7,640,24,4,Hires
Flash Off : Curs Off : Cls 0
Palette $F00,$FFF,$FF0
```

```
Rem *** Print headings...
Pen 1 : Paper 0
```

```
Locate 2,1 : Print "Screen X = "
Locate 20,1 : Print "Screen Y = "
Locate 38,1 : Print "Hard X = "
Locate 56,1 : Print "Hard Y = "
```

```
Pen 2 : Paper 0
```

```
Repeat
  Rem *** Move screen...
  Screen Display 7,128,SCRYPOS,,

  Rem *** Has up arrow key been pressed?
  If Key State(76)=-1
    SCRYPOS=SCRYPOS-1
  End If

  Rem *** Has down arrow key been pressed?
  If Key State(77)=-1
    SCRYPOS=SCRYPOS+1
  End If

  Screen SCRNO

  Rem *** Read mouse pointer positions...
  SCRX=X Screen(X Mouse)
  SCRY=Y Screen(Y Mouse)
  HRDX=X Mouse
  HRDY=Y Mouse

  Screen 7

  Locate 12,1 : Print SCRX;" "
  Locate 30,1 : Print SCRY;" "
  Locate 46,1 : Print HRDX;" "
  Locate 64,1 : Print HRDY;" "

  Wait Vbl
Until Key State(81)=-1 : Rem *** F2 to quit...

Screen Close 7
Amal On
Change Mouse 1
End If
End Proc
```



## High-score table

Every game player likes to feel that they achieved something, so a high-score table routine is a must for your AMOS games. And, surprise surprise, that's exactly what I've got for you below. It's fairly minimal, so don't expect any flash graphics – all it does is to handle the task of maintaining the high score table which is held within two arrays – HISCORE() and HISNAME\$(). Here's the high score code...



```
Rem *** Basic High Score Routine
Rem *** By Jason Holborn
```

```
Dim HISCORES(10),HISNAME$(10)
Global HISCORES(),HISNAME$()
```

```
Screen Open 0,320,256,2,Lowres
Flash Off : Palette $0,$FFF
```

```
For C=0 To 9
    Read HISCORES(C),HISNAME$(C)
Next C
```

```
Data 10000,"Jason Holborn"
Data 9000,"Georgina Brown"
Data 8000,"Barry Whitehouse"
Data 7000,"Rod Lawton"
Data 6000,"Dave Smithson"
Data 5000,"Marcus Dyson"
Data 4000,"Cliff Ramshaw"
Data 3000,"Dan Slingsby"
Data 2000,"Richard Vanner"
Data 1000,"AMOS Professional"
```

```
FMAT$=" ~~~~~ "
```

```
Do
    Input "Enter Score: ";SCORE
```

```
HISCORE[SCORE]
SUCCESS=Param
If SUCCESS=1
    Print
    For C=0 To 9
        Print Using FMAT$;HISNAME$(C),HISCORES(C)
    Next C
    Print
Else
    Print "*** Score too low!!!"
End If
Loop
```

Rem \*\*\* HiScore Table Routine

```
Procedure HISCORE[SCORE]
    If SCORE>HISCORES(9)
        Print "Congratulations!"
        Input "Enter name - ";NME$
        For C=9 To 0 Step -1
            If SCORE>HISCORES(C)
                POS=C
            End If
        Next C
        For C=8 To POS Step -1
            HISCORES(C+1)=HISCORES(C)
            HISNAME$(C+1)=HISNAME$(C)
        Next C
        HISCORES(POS)=SCORE
        HISNAME$(POS)=NME$
        RTURN=1
    Else
        RTURN=0
    End If
End Proc[RTURN]
```

## Bubble sort

Quite a few of the readers of Amiga Shopper and users of the bulletin board system '01 For Amiga' asked me to include a sort routine in the book that sorts an array filled with strings into alphabetical order. Well, being the helpful chap I am (!), here's an AMOS version of a 'bubble sort' routine I wrote in 'C' a few years back.

```
Rem *** String Array Bubble Sort routine
Rem *** Filename - BubbleSort.AMOS
```



```
ARRAYLEN=10 : Dim ARRAY$(ARRAYLEN)
Global ARRAY$( ), ARRAYLEN
```

```
Screen Open 0,640,256,2,Hires
Flash Off : Curs Off : Cls 0
Palette $0,$FFF
```

```
For C=0 To ARRAYLEN-1
    Read ARRAY$(C)
Next C
```

```
Data "Jason Holborn"
Data "Georgina Brown"
Data "Barry Whitehouse"
Data "Rod Lawton"
Data "Dave Smithson"
Data "Marcus Dyson"
Data "Cliff Ramshaw"
Data "Dan Slingsby"
Data "Richard Vanner"
Data "AMOS Professional"
```

```
Print "Here's the array in its unsorted form..."
Print
For C=0 To ARRAYLEN-1
    Print ARRAY$(C)
```

```
Next C
Print
Print "Press any key to sort it..."
Wait Key

_BUBBLESORT

Print
For C=0 To ARAYLEN-1
    Print ARAY$(C)
Next C
Print
Print "...and here's the sorted array!"
End

Procedure _BUBBLESORT
    FLAG=0

    Repeat
        FLAG=-1
        For C=0 To ARAYLEN-2
            If ARAY$(C)>ARAY$(C+1)
                FLAG=0
                TEMP$=ARAY$(C)
                ARAY$(C)=ARAY$(C+1)
                ARAY$(C+1)=TEMP$
            End If
        Next C
    Until FLAG=-1
End Proc
```



# Appendix B: Getting your game published

- Approaching a software house
- Stopping yourself from getting 'ripped off'
- Hiding your game's creator!
- The PD options – including shareware & licenseware

Let's face it, most people bought AMOS Basic to launch them on the way to programming fame and – hopefully – a little fortune chucked in for good measure. Whilst sending the odd listing in to magazines such as Amiga Shopper or Amiga Format, or even submitting a program for a coverdisk will undoubtedly get you the former (that's the fame bit!), not even the magazines themselves will try to pretend that you'll make a fortune from getting your listings printed within their illustrious pages. If you want to make real money from your programming effort, then you're going to have to release it as a commercial product.

Getting a game onto the shelves of WH Smiths isn't as easy as it may first appear, however. As many AMOS users have discovered, most software houses still hold a very dim view of any software written any form of Basic, let alone AMOS. Even if you were to write the ultimate arcade game that knocks the spots off of Team 17's 'Body Blows', 'Project X' or 'Alien Breed 2', chances are that most of the software houses you send the game to will reject it without even viewing it properly. The reason? – because it's written in Basic. It's a simple (and unavoidable) fact of life that software houses are particularly keen to distance themselves from programs written using so called 'game creators' (AMOS may be a powerful programming language, but most softies still tend to categorise AMOS as a 'game creator'), so getting an AMOS program published is no simple feat.

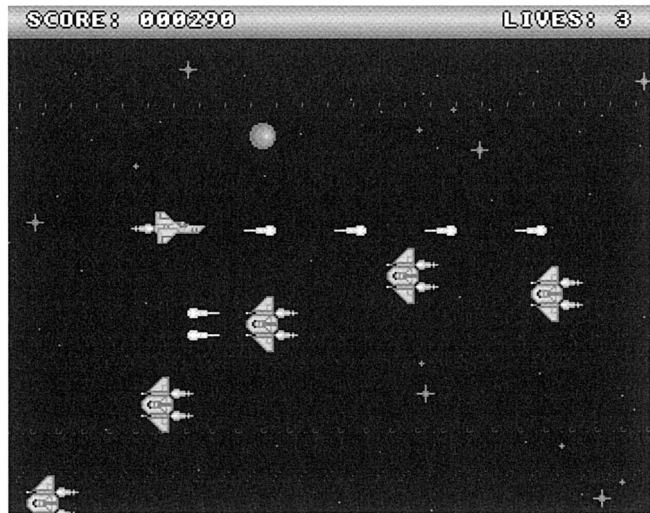


---

## Facing up to reality

So how do you go about getting an AMOS program published? Well, the first thing to do is to take a step backwards and ask yourself if the program genuinely is of commercial quality. OK, so we're all very proud of our own programs, but you must look at the situation realistically. Better still, get a couple of your friends to take a look at the program for you and take note of any suggestions that they make. If, for example, they say that the graphics are a bit naff or the soundtrack is annoying, then do something about it. The quality of gameplay is important too – if your friends find the controls frustrating or the screen update too slow, then don't go into a sulk and storm out of the room ranting something along the lines of how they couldn't do any better. Take note of what they say and act upon it!

*If you've ever wanted to make money from your AMOS creations, then read on...*



Quite a few 'serious' programs have been written in AMOS, but I'm afraid that, even if they are up and beyond the quality of current commercial equivalents, software houses won't touch AMOS applications. The simple fact of the matter is that serious users like their software to run under Intuition (the Amiga's windowing environment), so until Europress launch the fabled AMOS Professional Intuition extension, commercial applications written in AMOS are a definite no-no. The only exception to this rule is educational software – indeed, most educational titles released these days are written in AMOS!



**Create  
something  
new**

Another good idea is to take a look at the products currently out there in the marketplace. If you've just written a game that has already been done a hundred times before, then it's going to have to offer something dramatically new if people are going to shell out their hard-earned cash.

When I say dramatic, I'm not just talking about pretty graphics – where non-original games ideas are concerned, you've got to be talking major amounts of playability and addictiveness that will make game players (even those that might already have a game of the same type) buy your version! A good example of this (once again) Team 17's 'Project X' – let's face it, horizontal scrolling space shoot 'em ups are hardly leading-edge stuff!



One of the areas that lets most AMOS games down the most is that of graphic and audio presentation. But although you should try to make your game's graphics and sound track as attractive as possible, don't worry too much if your game isn't up to the same visual and aural standards as a commercial release. What you must consider is that software houses employ staff specifically for the task of designing game graphics and writing game soundtracks. These people – that are usually very talented anyway – spend their entire lives sat in front of DPaint and SoundTracker creating real audio visual extravaganzas. If a software house decides to market your game, they'll probably suggest that one of their graphic artists or musicians provides some new graphics or sound.

---

## AMOS in disguise!

If you're still totally convinced that your game is the best thing since sliced bread, the next thing that you absolutely must do is to disguise the fact that it was written in AMOS Basic. I know, we're all proud to be AMOS coders, but software houses don't give a hoot about such banner-waving, so save it for your fellow AMOS coders. The first thing you should do is to never (and I repeat NEVER) send your game in as nothing more than just source code and expect the software house to own a copy of AMOS. Not only will they not own AMOS, but it's doubtful whether your game will ever make it into the disk drive of one of their Amigas.



**Compile your  
game!**

Before you even think about submitting a game, you should always compile it. Even if you don't own the AMOS Compiler, hold onto your game until you've saved up enough cash to buy a copy. Not only will your game be far more professional (it will run from any disk without the need for a runtime system), but it will probably benefit from the extra spurt of speed anyway. Here's a couple of extra tips worth considering.

- 1 Although Europress would like you to advertise the fact that you used AMOS to produce your game, you're not bound by any legal obligation. Europress did originally insist that the AMOS logo was included on the title screens of all AMOS-produced software, but after realising the difficulties that AMOS users were encountering when trying to sell software, they modified these obligation a bit.

**Legal obligations**

You now don't have to advertise your program's origins at all, so the software house that you submit your game to need be none the wiser! Instead, all you have to do is to inform Europress (preferably in writing – mark your letters for the attention of Chris Payne) that you've written an AMOS game that will be sold commercially (note that this letter should only be sent after your game has been accepted). Chris and the rest of the Europress marketing team will keep this fact a secret for two months after your game has been released. After the two months have expired though, they'll have the right to advertise the fact as much as they like.

Don't worry about the software house getting miffed – if they've made bags of money out of your game, they'll be too busy counting it to worry! If the game is a flop though, chances are that Chris won't say a thing anyway.

- 2** It may be very handy for your own programs, but never use the AMOS file requester within your game. Not only is this a dead giveaway, it's not even a particularly good file requester. If you feel up to it, code your own.
- 3** Another way of giving the game away (if you'll pardon the pun) is to stick with the standard AMOS mouse pointer, which is very distinctive to say the least. If you want to change it to something less obvious, simply load the file 'mouse.abk' (you'll find it in your AMOS System folder) into the AMOS Object Editor and change the shape of the pointer yourself.
- 4** Use permanent memory banks as much as possible. There's nothing naffer than having all your graphics and sounds held on disk as separate files. Although this isn't as bad with assembler programs (you can encode them as you like), AMOS insists that pictures are stored in standard IFF format, so any Tom, Dick or Harry will be able to load your pictures. Using sprite and bob banks that are separate from the main program is another definite giveaway too.

## Making your move

Now that your game is in a form that's suitable for submission, make any final checks that are necessary to ensure that it doesn't have any bugs. Software houses are very quick to dismiss a program under any circumstances, so don't send them a program that is likely to crash. What you must consider is that the Software Manager (the guy who will check out your program) is regularly sent piles of submissions, all of which he must check through. If your program crashes after little more than a few minutes, he's unlikely to want to wait while it loads again.

Debugging a program before it is submitted is somewhat different to the sort of debugging you'd usually do. Put yourself in the position of someone who knows absolutely nothing about computers, let alone your own particular program. Better still, grab someone who you consider to be totally brain-dead (and no, I'm not available for this sort of thing!) and stick them down in front of your program. It's all too easy for someone who is very close to a programming project to miss what would seem to be a blindingly obvious bug to an outsider. Because you already know how to use your program, you're unlikely to start doing things with the program that you shouldn't – striking the wrong key, for example.



### Bug-checking

Documentation is also another very important factor to consider. Although your game may seem logical enough to you, the Software Manager who will be testing your program doesn't want to have to work out how to play the game for himself. It's therefore very important that you supply some form of documentation with your program. Don't go totally overboard here – the Software Manager doesn't want to know the politics behind why a lone spaceship from the outer galaxies is attacking a race of green slimy space pirates (even the people that buy your game probably won't be interested in this sort of information unless it is directly relevant to the gameplay).



### Clear documentation

Don't write too much, either – if the Software Manager has to wade through hundreds of pages of documentation just to find out how to load your program, I can guarantee you that your game won't even get near an Amiga disk drive! Just a page or two is more than enough and it should contain the following items.

### **Program Title**

The name of your program (e.g. 'Space Mutants From the Sun'). Don't worry too much about the title – the software house will probably change it anyway!

### **Machine**

What type of machine it runs on. Don't write down every type of Amiga under the sun (A500, A600, A600HD etc) – just write Amiga.

### **Memory**

The minimum amount of memory required to run the program. Don't go overboard with your memory requirements – most software houses will only accept games that run comfortably on a machine with no more than 1Mb. If your game requires something ridiculous like 4Mb, then the potential market for your game will be so small that it won't be worth the software house's time releasing it.

### **Description**

Keep this very brief – all the Software Manager needs is a very brief description of the game. Include such facts as the game type (shoot 'em up, 3D dungeon exploration game etc), a description of how the game is played and the structure of the game (how many levels it has, any extra power ups that the player can collect etc).

### **Controls**

What controls does the game use? If the game is keyboard-controlled, give a full list of all the keys.

### **Loading Instructions**

Always make your game autobooting so that the Software Manager can simply turn on his Amiga, insert the game disk and the game will load. If any setting up is required, document these in concise detail.

---

## **Legal eagle**

With all this done, you're almost ready to send your game off. Before you do so, however, protect yourself by posting off a copy of the game to yourself and when it arrives DO NOT OPEN IT. This package will be



**Protect  
yourself!**

used to prove when the game was written and you should therefore place it in the hands of either a solicitor or (cheaper still) take it along to your bank and ask them to hold onto it. Either way, you'll be asked to pay a retaining fee but this is well worth paying for if – in the unlikely event you are ripped off – you have to prove that the game (or, in the case of original games, the game *idea*) was yours. Always keep copies of all letters that you send to a software house and, when you finally do send off your game, ask the post office for a 'proof of postage' certificate.

---

## Get noticed

So how do you get your game noticed? Well, one of the most common methods used in the music industry is to send in a tape that is brightly coloured. This works equally well in the computer industry too – because most people send in boring blue or black disks, sending in a disk that is brightly coloured (red, green and yellow disks are available) will make your disk stand out from the rest.



**Stay cool**

Finally, you need to establish what software houses would be best suited to market your particular game. If, for example, you've written an arcade game, there's little point in trying to sell your game to a company that specialises in strategy games. Always go for a software house that has a proven track record in marketing the same type of game as the one that you have written. Address your game to the 'Software Manager' and then sit back and wait. Don't keep hassling a software house if you haven't received a reply from them within a month – because of the large number of disks that they receive, they're unlikely to find time to look at your disk within the first couple of weeks.

Feel free to send your game to several publishers, but don't tell them that other rival publishers have also received a copy. If several companies to express an interest, then you can tell them about their rivals. If they are genuinely interested in your game, they'll be happy to compete for it. Don't push your luck, though – as they say, a bird in the hand is worth two in the bush!

## The PD option

If your game isn't quite up to the sort of standards that commercial software houses demand, then why not send your game to a PD software library? OK, so PD distribution is hardly going to earn you any money, but it's the recognition that counts – just ask any commercial games programmer. Most games programmers started this way. Take Team 17, for example. Martyn Brown (MD of Team 17 and all round bitter-drinking good guy) used to work for the PD software house '17 Bit Software' (hence the name 'Team 17') and it was through his exposure to PD demos and games that he was able to find a team of programmers, graphic artists and musicians. All of the closed group of programmers that work for Team 17 caught Martyn's eye due to their PD efforts. And, as they old saying goes, if they can do it then so can you!

It is possible to make a little money from public domain if the thought of international recognition isn't enough for you. PD software comes in a number of different flavours ranging from 'Freeware' (anyone can copy or sell your program without your consent) to 'Beerware' (the players of your game send you a six-pack!) and – most interesting of all – 'Shareware'. Shareware is a system where you put some sort of message into your game that kindly asks anyone who buys a disk containing your program to send you a donation if they like your program (usually around £5-10).



**Shareware**

Obviously, shareware is dependent on the honesty of the people that buy your program. But don't knock it – Jeff Minter (he of 'LlamaSoft' fame), for example, recently stopped issuing his games as commercial products and put them into the PD libraries as shareware products. By promising anyone who sent him a crispy £5 note that he would send them his next game, Jeff received well over £10,000 in shareware donations!

If you want to ensure that a greater proportion of the people that use your program donate a little money, then it's not a bad idea to limit the shareware release in some way, to give those who use your program an incentive to donate their hard-earned cash. If you were writing a game, for example, you could limit the shareware version to only a couple of screens (a 'taster' if you like) and then promise to send anyone who

donates money a version of the game with a lot more screens! This is exactly the approach that many shareware authors adopt and it certainly seems to work.



### Licenseware

A relatively new PD distribution system is 'Licenseware', formulated by my old friend Sandra Sharkey at Deja Vu software. The basic idea is that if your game is good enough, Sandra will distribute it to a number of PD libraries that are registered to sell licenseware disks. What's more, the PD libraries in question have an obligation to charge a fixed price of £3 for all licenseware programs, of which £1 is paid to you, the programmer. OK, so it doesn't sound like a lot of money, but a successful licenseware program could easily earn the programmer a four-digit figure!

# Appendix C: Where to go from here

- 'The AMOS Club'
- 'Totally AMOS'
- Magazines
- Bulletin boards



**O**ver the last 380 or so pages, I've tried to cover just about every programming concept you need to know in order to write arcade games and demos in AMOS, but even the considerable amount of routines, tricks, hints and techniques we've discussed form only a basic knowledge of the games programmer's art. Sure, you should now know enough about AMOS to churn out AMOS games by the bucketful, but even I won't try to delude you that the games programming techniques that we've covered are going to launch you to international stardom. Don't get me wrong – if you've fully understood and taken in every single programming technique covered within these hallowed pages, then you're now one of the AMOS programming elite – but it's down to you to hone your skills from here on.

In many ways, 'Ultimate AMOS' has given you something of a head-start – when I was learning to program games many years ago, there was nothing like 'Ultimate AMOS' around. About the closest us veteran programmers ever got to a games programming tutorial was a tacky listing in 'Home Computer Weekly' (anyone remember that esteemed publication?) that had a little 'X' firing asterisks at aliens that looked suspiciously like the letter 'M'! Of course the hardware was no where near as powerful as the Amigas we have today – 1K of RAM and a black and white low resolution screen was the best that money could buy!

As Dr. Ruth would no doubt agree, the best way to become better at anything is to keep trying and to keep your mind open to new ideas and techniques. If you're writing a game that uses a routine that you've never tried writing before, don't just use the first version that comes into your head – keep thinking all the time of new ideas that could possibly enhance the routine to make it leaner and meaner than ever before. Many amateur programmers seem to live by the old saying 'if it works, don't touch it' but as a games programmer you must strive to make your code as fast as possible. Every time you squeeze that extra bit of performance out of your code, that's space enough for you to add an extra feature to your game...

There are many aspects of AMOS that we haven't even covered within 'Ultimate AMOS' that I would have loved to have included – AMOS Professional's 'Interface' language, AMOS 3D, how to program 'serious'

applications like databases and word processors etc – but there just wasn't the space. When I started this book, some very hard decisions had to be made. Many previous AMOS books had tried to cram in everything but the kitchen sink, but they've lacked the sort of detail that AMOS programmers demand. By concentrating on games only, however, I've been able to go to town (in a manner of speaking) with a very high level of detail and I hope that you've found the book not only very useful, but rewarding too.

---

## The AMOS Club

This may be the end of the book, but that doesn't mean that you're on your own from here on. There are a number of AMOS user groups and clubs of which the most popular is undoubtedly Aaron Fothergill's imaginatively named 'AMOS Club' which is the only AMOS club supported by Europress Software, the guys behind AMOS. Aaron (who, incidentally, is the programmer of the 'Tome' and 'CText' extensions as well as being head-honcho at Shadow Software) runs a telephone support line for AMOS Club members and there's also a regular newsletter and disk magazine to keep you interested through those long winter nights. The AMOS Club can be contacted at the following address.

**The AMOS Club  
1 Lower Moor  
Whiddon Valley  
Barnstaple  
North Devon  
EX32 8NW**

---

## Totally AMOS

Another valuable source of AMOS advice and ideas that I must mention is 'Totally AMOS', a disk-based magazine produced by Len and Anne Tucker, two very well known AMOS personalities. New issues of Totally AMOS are released every two months and they contain a wealth of AMOS-related source code, tutorials, reviews, programming tips, competitions and useful graphic and sound files which are contributed to the magazine by Totally AMOS members, amongst them some of the

best in the business. A year's subscription to Totally AMOS costs £18 but this also entitles you to a 10% discount on the purchase of disks from the AMOS PD Library which Len and Anne also just happen to run. Totally AMOS and the AMOS PD Library is available from the following address.

**Totally AMOS  
1 Penmynydd Road  
Penlan  
Swansea  
SA5 7EH**

---

## Magazines

Amiga Shopper and Amiga Format are good sources of AMOS news and information and both just happen to be published by the same company – Future Publishing – that has brought you this wondrous volume. In particular, I'm sure you'll be interested in the 3-page AMOS column in Amiga Shopper.

Amiga Shopper also runs a very useful 20-page 'Amiga Answers' section which aims to answer the technical questions that other magazines avoid. The 'Answers Panel' consists of some of the most respected Amiga journos in the business including Mark Smiddy, Gary Whiteley, Jeff Walker and my humble self. Every month the Answers Panel wades through technical questions on just about every Amiga-related subject. If you have an AMOS query, then why not buy yourself a copy of Amiga Shopper and find out how you can get the Answers Panel working for you!

---

## Bulletin boards

If you're lucky enough to own a modem, then megabytes of AMOS source code and the answers to your AMOS queries are only a phone call away. Quite a few bulletin board systems (BBS's) are frequented by AMOS users and experts alike. If you've got a problem that you just can't work out for yourself, all you have to do is to log onto a bulletin board, post up a message detailing the problem that you've got and the

chances are that you'll get a reply within a day or so. I myself 'hang out' on a number of BBS' including possibly the UK's finest Amiga-only BBS, 01 For Amiga, which is run by my old friend Tony Miller.

01 For Amiga also offers an absolutely enormous file area which includes a section dedicated to AMOS. Within this file area you'll find AMOS extensions, demos, the latest AMOS 'updater' disks and more source code than you could shake a stick at. Here's a list of BBS's that support AMOS.

<b>01 For Amiga</b>	<b>071 377 1358</b>
<b>AMOS BBS</b>	<b>010 325 842 2433 (Belgium)</b>
<b>The End Zone</b>	<b>0524 752245</b>
<b>Amiga Pond</b>	<b>051 547 3245</b>
<b>Cheam Amiga</b>	<b>081 644 8714</b>
<b>CIX (Subscription only)</b>	<b>081 390 1244</b>



# Index

Includes AMOS/AMAL commands and functions

<b>A</b>	
	AGA (Advanced Graphics Architecture) .....53
	AGA palette .....64
	AGA screen modes .....57
	AGA support.....14
	AMAL.....223, 275
	Beyond 16 channels .....238
	'Embedding' AMAL code.....226
	'Amal' AMAL command.....228
	AMAL Editor .....225
	AMAL functions.....235
	AMAL instruction set .....232
	AMAL principles.....225
	AMAL registers.....229
	Special AMAL registers .....231
	'AMAS 2' .....17, 254
	'AMOS 1.35'.....6
	'AMOS 3D'.....9
	'AMOS Compiler' .....9, 386
	AMOS Editor .....22, 23
	AMOS Monitor .....34
	'AMOS Pro Compiler' .....10, 275
	'AMOS Professional' .....7
	'AMOS TOME' Extension .....11, 100
	AMOS 'updater' disk.....6
	ARexx ports .....8
	Adventure games .....342
	Agnus chip .....104
	Alice chip .....104
	'Amiga Format' magazine .....396
	'Amiga Shopper' magazine .....396

'Amreg()' AMAL function .....	230
'And' logical operator .....	177
Animation – Changing the frame sequence .....	151
Slowing down animations .....	147
'Anim' AMAL command .....	233
'Anim Off' command .....	151
'Anim On' command .....	151
'Anim' command .....	149
Animation Editor .....	169
Array .....	172
Attack waves .....	200
'AudioMaster' .....	13
'AutoBack' command .....	117, 218, 276

## B

Banks .....	128
'Bell' effect .....	253
Bit pattern .....	176
Blitter .....	76
'Bob Clear' command .....	216
'Bob Col()' AMAL function .....	235
'Bob Col()' function .....	158
'Bob Redraw' command .....	216
'Bob Update Off' command .....	216
'Bob Update On' command .....	216
'Bob' command .....	138
Bob redrawing .....	276
'Bobsprite Col()' function .....	159
'Boom' effect .....	253
Bubble sort .....	380
Bulletin boards .....	396



<b>C</b>	'CTEXT' .....	12
	'Channel' AMAL command .....	227
	'Channel' command.....	150
	Circular attack waves.....	207
	Co-ordinate finder .....	375
	Code comments .....	44
	Code indentation .....	44
	'Col()' AMAL function .....	236
	'Col()' function.....	160
	Collision detection .....	157, 294
	'Colour' command.....	63, 64
	Colour components .....	63
	Computed sprites.....	136, 142
	Continuous scrolling.....	92
	Copper .....	104, 142
	Copper bar .....	112
	Copper list .....	104
	'Cos()' function.....	193
<b>D</b>		
	'D-Sam' extension .....	12, 263
	Data structure .....	172
	DeInterlacer .....	57
	Debugging .....	34
	'Def Scroll' command.....	82
	'Degree' command.....	194, 201
	'Deluxe Paint'.....	16
	Denise chip .....	104
	'Dim Array' command.....	172
	Direct mode .....	23
	Documentation .....	388

---

	'Double Buffer' command .....	213
	Double buffering .....	116
	'Dual Playfield' command .....	87
	'Dungeon Master'-type games .....	317
<b>E</b>	'Easy AMOS' .....	5
	Editor menu .....	24
	Editor options .....	24
	ECS (Enhanced Chip Set) .....	53
	'Erase' command .....	131
	Extra Half Brite mode .....	54
<b>F</b>	'Fire()' function .....	179
	Firing missiles .....	209
	'For... To... Next' AMAL command .....	234
<b>G</b>	'Get Sprite Palette' command .....	138
	Global AMAL registers .....	230
	'Global' command .....	48
	Global variable .....	47
<b>H</b>	HAM-8 mode .....	53, 57
	Hardware Scrolling .....	75, 76, 243
	Hardware Sprites .....	134
	'Hex\$( )' function .....	64
	High-score table .....	378
	'Hot Spot' command .....	155
	Hot spots .....	123
	'Hrev()' function .....	152
	'Hzone()' function .....	126

<b>I</b>	IFF ANIM format.....8
	IFF graphics standard.....70
	Icon bank.....121
	Icon commands.....120
	'If... Jump' AMAL command .....234
	'Inkey\$( )' function.....181
	'Input' command .....181
	Interface .....8
	Interlace mode .....54
	Interrupt-driven animations.....149
	Intuition extension.....13, 385
<b>J</b>	'Jdown()' function .....179
	'Jleft()' function .....179
	'Joy(1)' function .....177
	'Joy0' AMAL function.....236
	'Joy1' AMAL function.....236
	Joystick .....175, 236
	'Jright()' function .....179
	'Jump' AMAL command .....234
	'Jup()' function .....179
<b>K</b>	'Key Shift()' function .....182
	'Key State()' function .....183
	'Key1()' AMAL function.....236
	'Key2()' AMAL function.....236
	Keyboard control.....181
	Keyboard shortcuts .....33

<b>L</b>	'Led Off' command.....	256
	'Led On' command.....	256
	Left mouse button .....	236
	'Let' AMAL command .....	234
	Licenseware.....	392
	Linear programming .....	43
	Lisa chip .....	104
	'List Bank' command.....	130
	'Load IFF' command.....	71
	Loading and saving screens .....	70
	Local AMAL registers.....	230
	Local variables .....	47
	'Logic()' function .....	118
	Logical screen.....	116
	Low resolution.....	54
	<b>M</b>	'MED' .....
Main game loop.....		272
'Make Mask' command.....		158
Mandelbrot generator .....		362
'Map Bottom X,Y' TOME command .....		102
'Map Do X,Y' TOME command.....		101
'Map Fall TILE' TOME command .....		102
'Map Left X,Y' TOME command .....		102
'Map Plot TILE,X,Y' TOMR command.....		102
'Map Right X,Y' TOME command .....		102
'Map Top X,Y' TOME command .....		102
'Map View X1,Y1 To X2,Y2' TOME command.....		101
Mask .....		158
Maths functions .....		193

Mathtrans.library .....	193
Maze games .....	294
Medium resolution .....	54
Modular programming.....	43
'Mouse Zone' command .....	124
'Move' AMAL command .....	233
Music.....	252
Music modules .....	257

## N

NTSC.....	55, 74
-----------	--------

## O

Objects.....	134
360-degree movement.....	193
Advanced object movement.....	192
Animating objects .....	146
Bouncing a bob .....	190
Creating sprites and bobs .....	136
Displaying an object.....	138
'Flipping' objects.....	152
Interactive object control.....	175
'Jumping' movement .....	196
Moving an object .....	140
Restricting object movement .....	185
Object Editor .....	137, 164
On line help .....	33
Opening screens .....	58
Optimising code .....	213, 275
Overscan .....	56

<b>P</b>	PAL .....	74
	'Palette' command .....	62, 64
	Parallax scrolling .....	86
	'Param' command .....	47
	Parser .....	343
	'Paste Icon' command .....	122
	Paula chip .....	252
	'Pause' AMAL command .....	235
	'Physic()' function .....	118
	Physical screens .....	116
	Platform games .....	322
	Procedures .....	46
	Pseudo code .....	41
	Public domain .....	391
	<b>R</b>	'REM' command .....
Radians .....		201
'Rain()' function .....		106, 110
Rainbows .....		105
Defining a rainbow .....		106
Rainbow animation .....		248
'Rainbow' command .....		106, 107
Raster beam .....		114
'Reserve Zone' command .....		124
'Rev()' function .....		152
Right mouse button .....		236
Runtime error checking .....		276
Runtime system .....		275

<b>S</b>	
	'Sam Play' command.....255
	Sample Bank Maker .....254, 264
	Sample bank .....254
	'Save IFF' command.....71
	Scan codes .....181, 184
	Scan lines .....110
	'ScanCode()' function .....182
	Screens .....52
	Resizing and positioning screens .....65
	'Screen Close' command.....61
	'Screen Copy' command .....84, 89
	'Screen Display' command .....65
	'Screen Offset' command .....77, 78
	'Screen Open' command .....59
	'Screen Swap' command .....117, 218
	Screen zones .....123, 323
	Screen combinations .....54
	'Screen' command .....61
	Screen compaction.....128, 129
	Screen icons .....120
	Screen management.....60
	Screen number.....58
	Screen palettes .....62
	Screen refresh rate.....272
	Screen scrolling .....74
	Screen synchronisation .....74, 113
	Screen updating .....140
	'Scroll' command.....82, 83

Scroll types .....	75
Using hardware scrolling .....	77
Using software scrolling .....	81
'Set Bob' command.....	217
'Set Rainbow' command .....	106
'Set Zone' command .....	124
'Shared' command .....	48
Shareware .....	391
'Shoot' effect.....	253
Shoot 'em ups .....	280
'Sin()' function.....	193
'Snake' attack waves.....	204
Software scrolling.....	75, 76
Software sprites (bobs).....	136
'Sound Tracker' .....	17, 258
Sound Tracker 'module' format .....	258
Sound effects.....	252
Sound filter .....	256
Sound sampling.....	17, 253
'Spack' command .....	130
'Splerge' effect.....	366
Sprite bank .....	136
'Sprite Col()' AMAL function.....	235
'Sprite Col()' function .....	159
Sprite Editor.....	137
'Sprite Update Off' command.....	143
'Sprite Update' command .....	143
'Sprite' command.....	138
Sprite playfield .....	135
'Spritebob Col()' function .....	159
Sprites 600 disk.....	146



	Starfield .....	369, 371
	'Step' variable.....	296
	'StereoMaster' .....	17
	Subroutines.....	43
	SuperBitmap .....	77
	SuperHiRes .....	57
	'Synchro' command .....	235, 238
	'Synchro Off' command .....	238
	Synthetic instruments.....	258
<b>T</b>		
	'TechnoSound Turbo' .....	254
	'The AMOS Club' .....	395
	'The Tutor'.....	34
	'Tome Map Editor'.....	12, 100
	'Totally AMOS' .....	395
	'Track Load' command.....	259
	'Track Loop On' command.....	260
	'Track Play' command .....	259
	'Track Stop' command .....	260
<b>U</b>		
	'Unpack' command .....	131
<b>V</b>		
	VGA .....	57
	'VU()' AMAL function .....	236
	VU meter .....	236, 260
	Vertical blanking period.....	114
	Viewport .....	66
	'Virtual' sprites.....	142
	'Vrev()' function.....	152
	'Vumeter()' function .....	260

<b>W</b>	'Wait Vbl' command.....	75, 114, 272
<b>X</b>	'X Hard()' function .....	139
	'X Screen()' function .....	139
	'XHard()' AMAL function .....	237
	'XMouse()' AMAL function.....	237
	'XScreen()' AMAL function.....	237
	Xor logical operator .....	153
<b>Y</b>	'Y Hard()' function .....	139
	'Y Screen()' function .....	139
	'YHard()' AMAL function .....	237
	'YMouse()' AMAL function.....	237
	'YScreen()' AMAL function.....	237
<b>Z</b>	'Z()' AMAL function .....	237
	Zone() function .....	126, 324

## THE SMALL PRINT

AMOS is the Amiga's premier games creation package. Since its launch it has been continually improved and updated, and third-party suppliers have provided extensions and utilities to further enhance AMOS's amazing abilities.

Ultimate AMOS covers every version of AMOS released, right up until the present day and the launch of the AMOS Pro Compiler – the long-awaited AMOS Pro add-on which converts AMOS Pro programs into fast-running machine code.

AMOS is now established as the Amiga's premier 'amateur' programming language.

The author of Ultimate AMOS, Jason Holborn, is the UK's leading AMOS journalist, and writes for Amiga Shopper and Amiga Format.

**AMIGA**  
FORMAT

## Inside...

- What AMOS can do
- AMOS variants and extensions
- How AMOS works
- Program planning
- Handling screens
- Hardware and software scrolling
- Handling sprites and 'bobs'
- Incorporating sound and music
- Using AMAL

**Plus:** different game types and how to program them, how to produce stand-alone programs and where to go for more advice, help and information.

Includes dozens of handy routines for incorporating into your own programs and four working AMOS games.



Your guarantee  
of value



**£19.95**

This work is licensed under the  
Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) License.

To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>  
or send a letter to

Creative Commons,  
PO Box 1866, Mountain View,  
CA 94042, USA.

Copyright 1994 Jason Holborn (Content)

Copyright 1994 Future Publishing (Layout & Design)

Released under CC BY-SA 4.0 2019