



Director MX 2004 Games

Game Development with Macromedia Director



Director MX 2004 Games

Thanks to Mum

Director MX 2004 Games

Game development with Director

Nik Lever



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON • NEW YORK • OXFORD
PARIS • SAN DIEGO • SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Focal Press is an imprint of Elsevier



Focal Press
An imprint of Elsevier
Linacre House, Jordan Hill, Oxford OX2 8DP
30 Corporate Drive, Burlington MA 01803

First published 2005

Copyright © 2005, Nik Lever. All rights reserved

The right of Nik Lever to be identified as the author of this work
has been asserted in accordance with the Copyright, Designs and
Patents Act 1988

No part of this publication may be reproduced in any material form (including
photocopying or storing in any medium by electronic means and whether
or not transiently or incidentally to some other use of this publication) without
the written permission of the copyright holder except in accordance with the
provisions of the Copyright, Designs and Patents Act 1988 or under the terms of
a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road,
London, England W1T 4LP. Applications for the copyright holder's written
permission to reproduce any part of this publication should be addressed
to the publisher

Permissions may be sought directly from Elsevier's Science and Technology Rights
Department in Oxford, UK: phone: (+44) (0) 1865 843830; fax: (+44) (0) 1865 853333;
e-mail: permissions@elsevier.co.uk. You may also complete your request on-line via the
Elsevier homepage (www.elsevier.com), by selecting 'Customer Support'
and then 'Obtaining Permissions'

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

Library of Congress Cataloguing in Publication Data

A catalogue record for this book is available from the Library of Congress

ISBN 0 240 51949 3

For information on all Focal Press publications visit our website at:
www.focalpress.com

Typeset by Newgen Imaging Systems (P) Ltd, Chennai, India

Printed and bound in Great Britain

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

Contents at a glance

<i>Introduction</i>	<i>Learn to write Lingo or JavaScript and have fun doing it!</i>	xi
Chapter 1	Your first game	1
Section 1	Layout	21
Chapter 2	Images and computers	23
Chapter 3	The walk cycle	34
Chapter 4	Background art	46
Chapter 5	Using computer-generated imagery programs to create animation	56
Section 2	Scripting	65
Chapter 6	So what is a variable?	67
Chapter 7	In tip-top condition	86
Chapter 8	Using loops	109
Chapter 9	Keep it modular	116
Chapter 10	Debugging	129
Chapter 11	Integrating with Flash	140
Section 3	Putting it into practice	149
Chapter 12	Kids' stuff	151
Chapter 13	Board games	164

Chapter 14	Quizzes	182
Chapter 15	Platformers	201
Section 4	The third dimension	221
Chapter 16	Creating low-polygon characters	223
Chapter 17	3D basics	241
Chapter 18	What's in a w3d file?	251
Chapter 19	3D techniques	274
Chapter 20	Using Havok	288
Appendix A	Maths for games	302
Appendix B	Links	315
Bibliography		323
<i>Index</i>		325

Contents in summary

Introduction: Learn to write Lingo or JavaScript and have fun doing it! **xi**

How to get the best from the included CD. Where to look for further information on the web. A very brief guided tour of the Director MX interface.

Chapter 1: Your first game **1**

Jumping head first into a tutorial-based ‘getting started’. In this chapter the reader is guided through the process of creating a simple game. In a tutorial format the user is taken through creating artwork in Director and then adding some limited interactivity. The basics of using Director behaviors are presented along with keeping track of the board state in a way that the computer can understand.

Section 1: Layout **21**

The first section looks at creating the artwork and animation that will feature in the games we create.

Chapter 2: Images and computers **23**

Director is the perfect application to use for fast-moving sprite graphics games. But first we have to create the images that our sprite requires. In this chapter we look at some guidelines when creating images for a game.

Chapter 3: The walk cycle **34**

Every action that your character does will link back to a standard position and many will work in a loop. In this chapter we look at one of the most difficult areas of animation—the walk cycle—and using 10 simple rules we make the process much easier for the beginner.

Chapter 4: Background art **46**

Your animated characters will need to appear in some context. In this chapter we look at creating backgrounds to act as a setting for the action that takes place in your games.

Chapter 5: Using computer-generated imagery programs to create animation **56**

Animation is a time-consuming process. With a 3D animation program a single designer can make a set of sprite animations in a day. In this chapter we look at doing this with Lightwave 3D.

Section 2: Scripting**65**

Having developed the animation we now look at how to add interactivity to this art using Lingo or JavaScript.

Chapter 6: So what is a variable?**67**

After the first blistering introduction to scripting in Chapter 1, we now take a leisurely tour through the basics of programming, starting with variable types. In this chapter we look at the differences between integers, floats and strings. It is all presented using a cloned animated sprite. One sprite moves using integers, the other with floats. Key press detection adds data to a string variable. The user can then replay the animation as stored in the string using string slicing.

Chapter 7: In tip-top condition**86**

In this chapter we look at using conditional code using the marvelous ‘if’ statement. We also look at how we can extend the flexibility of the condition using Boolean logic. The concepts are illustrated using a switchboard display. By clicking on the switches we can light various lights on the light board.

Chapter 8: Using loops**109**

No programming language would be complete without the ability to repeat sections of code. Games regularly need to run a section of code several times. In this chapter we look at creating sections of code that repeat, how to jump out of the code if a condition is met, using repeats that run at least once and initializing the data used in a loop.

Chapter 9: Keep it modular**116**

Programming can be made very hard or very easy. Good structure makes it so much easier. We look at how to use custom functions to make sure that you only have to change your code in one place to have a global effect. We look at using functions to change the value of certain variables. We look at the difference between local and global variables.

Chapter 10: Debugging**129**

Nobody gets it right first time; in this chapter we look at how to get a program working when errors are occurring.

Chapter 11: Integrating with Flash**140**

Flash is a very useful tool to display dynamic game information. With the ability to embed fonts and generate complex on-screen effects, it extends Director in a very effective way. In this chapter we look at integrating Flash with Director.

Section 3: Putting it into practice **149**

You know your stuff so how about using this knowledge to make some games?

Chapter 12: Kids' stuff **151**

Director has often been used to create fun activities for young children. In this chapter we look at creating several of these and it forms a gentle introduction to the more complex scripting of the following chapters.

Chapter 13: Board games **164**

A board game involves a legal move generator and a mini-max search routine for a good computer move. In this chapter we look at the strategies involved in presenting some classic board games in Director form.

Chapter 14: Quizzes **182**

While examining how to present an on-line quiz we explore databases, server-side scripting and Flash UI components. Many of the lessons learnt in this chapter apply equally to creating registered users and high-score tables.

Chapter 15: Platformers **201**

Creating a platform-based game is quite a challenge, but by this stage in the book the reader has all the skills necessary. In this chapter we look at the problems of game world and screen space calculations. Platformers need to create sprites dynamically; we look at creating content on the fly.

Section 4: The third dimension **221**

The best thing about Director MX 2004 for the game developer is the 3D features. The final section takes the reader through the information needed to create their own 3D game.

Chapter 16: Creating low-polygon characters **223**

Using Lightwave 3D the reader is shown how to model a 3D character, how to bone the character, how to animate the character and finally lots of top tips to avoid problems when exporting.

Chapter 17: 3D basics **241**

Understanding the 3D environment can be hard for some developers. In this chapter we introduce 3D concepts and look at moving simple objects under keyboard control.

Chapter 18: What's in a w3d file? **251**

Director MX accesses 3D assets via a file format called w3d. This file contains textures, shaders, lights, cameras, models, model resources and motion resources. In this chapter we look at each of these elements and how they can be altered using Lingo.

Chapter 19: 3D techniques 274

In this chapter we look at useful techniques that you can adopt in many of your games: camera control and basic collision testing along with terrain alignment methods.

Chapter 20: Using Havok 288

Havok is an exceptionally powerful rigid body physics simulation system that comes with Director MX. In this chapter we look at how to control a physics simulation in the form of a snowboarding game. The chapter also introduces tips on character animation and more sophisticated collision testing.

Appendix A: Maths for games 302

When developing games a certain amount of mathematical knowledge is extremely useful. This appendix explains in nontechnical language how trigonometry, vectors and matrices can be useful when developing your games.

Appendix B: Links 315

A useful list of further information and inspiration available on the Internet.

Bibliography 323**Index** 325

Introduction: Learn to write Lingo or JavaScript and have fun doing it!

Director MX 2004 provides the perfect platform to create fun games for CD or Internet distribution. This book takes the reader through the entire process from creating the art and animation for these games, through to programming them ready for users across the world to enjoy the results. The book is split into four sections. Section 1 introduces the new user to basics of creating the animation art that your games will need, whether they are 2D or 3D. Section 2 takes the reader on a guided tour of using scripting; Director MX 2004 offers the programmer two scripting options, Lingo or JavaScript. Section 2 introduces the reader to both these options allowing them to decide which they prefer. The emphasis is on explaining computer programming to readers from a design background, no previous computer programming is assumed. In Section 3 lots of practical examples are offered of specific games, all the source code is provided in Director MX 2004 format on the included CD. Finally in Section 4 we look at moving into the third dimension. Director 8.5 first allowed the developer to create games that have the PlayStation appeal of 3D. Although 3D games are more demanding of the programmer, Director makes the move into 3D as painless as possible and the results speak for themselves. The last part of the book includes two appendices covering topics that do not fall naturally into the main text.

Using the CD

You can browse the folders on the CD. Each chapter and appendix has a unique folder in the 'Examples' folder. All the projects and source can be found here. To use a project file, open the file directly from the CD in Director MX 2004 and then save it to your local hard drive if you intend to make changes.

Who is the author?

It all started with a ZX81, a hobbyist computer released in the UK in 1981. The ZX81 was soon replaced by a Sinclair Spectrum, an amazing computer from the same company. The Sinclair Spectrum boasted an amazing 48 K of memory, and an eight-color display. From those early days the author was smitten, writing code became something of an obsession. Having graduated as a Graphic Designer in 1980 he had already started work as a professional animator when the interest in computers surfaced. All too soon, his computer hobby merged with professional life, the first example being a computer-controlled rostrum stand, a camera for filming 2D animation. Much more recently he has been producing CD-ROMs and web-based multi-media productions for clients including Cartoon Network, Kellogg's, Coca-Cola, BBC, Sekonda and Polydor.

The first interactive web-based work used Java, but the latest work tends to be all Flash or Director based.

Check out www.catalystpics.co.uk to see the author's handywork or www.niklever.net/director to see the web site for this book.

Who is this book for?

Designers

If you are a designer who has worked with Flash or Director and want to go further, you will find lots of interest here. You will learn about applying your creative skills to the many stages of game production. If you have never played with Director then fear not, you will be guided through the art, animation and programming involved in creating sophisticated games.

Animators

Perhaps you have created the sprites for other people's games and always meant to look into how to create your own; in this book you will learn how. You will see how to use Director to create smooth animation and then how to add interactive functionality to your game using scripting. Even if you have never written a line of code before then you will find the tutorial style easy to follow.

Web developers

If you haven't started writing Director code then shame on you, start today. Director offers a sophisticated development environment with good tools for the production of highly dynamic interactive activities. In this book you will learn how to use Director to the best by creating animation and then adding interactivity, producing exciting and dynamic Internet content.

Students

The Internet is rapidly turning into a rich source of employment for visually literate students. If your skills also included adding the code then you will be in demand. In this book you learn all the skills necessary to create highly dynamic web content.

So what are you waiting for?

Boot up your computer, go straight to Director MX 2004 and turn the page where you are going to create your first game in Director!

1 Your first game

Rather alarmingly it is now over 20 years since I wrote my first bit of code using a Sinclair Spectrum computer and Sinclair Basic.

In 1980, in the UK, Clive Sinclair, an entrepreneurial inventor, advertised for sale a very simple computer. The ZX80 connected to the TV set and enjoyed 1 K of RAM. Each machine came with a simple programming language, Beginners All-purpose Symbolic Instruction Code (BASIC). This early machine started a revolution of young bedroom and garage programmers. Some of these early starters went on to become very successful in the games industry. Sinclair's inexpensive computers, including the Spectrum, are one of the main reasons why the UK has an internationally respected games industry.

Animation was for several years my main motivation. I still get a buzz from making things move. Although initially unconnected, programming shares many similarities with animation. It is a creative pursuit. If you come to programming from an arts background then you may find this doubtful. Surely creative implies the visual, written or musical arts. But programming is not a case of one solution to a problem, there are many solutions and the path you take to the solution is where the 'art' comes in. I got such a buzz from seeing my first programs work that I now spend rather more of my time writing code than creating animation artwork. In this chapter we are going to rush headlong into creating a game. Not a massive multi-player 3D first-person extravaganza; we may have to wait for day two for that! No, the game we will produce will be familiar to anyone who has ever played a computer game. It is a very simple version of 'Pong'. On the way we will discover the Director interface. You will learn how to create simple artwork with Director, where to find all those windows and what they do, how to add a little code and how to test your work. So without further hesitation let's get going. I strongly advise reading this chapter while using your computer. You will get much more from it if you enter the code and follow the tutorial throughout rather than just dissecting the final program.

Creating simple artwork using Director

Figure 1.1 shows the basic Director interface. The biggest area is used for the 'Stage'. This is where we put the imagery that the user will see. Below this in the diagram is a grid; this is the 'Score'. In Director a movie is broken down using time segments and layers. A single column in the Score represents one time segment and a row represents a layer of graphics or a sprite. So how long is a time segment? By default a time segment is one thirtieth of a second. The user can alter the duration of a time segment by opening the 'Control' window. To open a new window select 'Window' from the main menu bar and then choose which window to open from the drop-down menu; in this case select 'Control Panel'. The window should look like that shown in Figure 1.2.

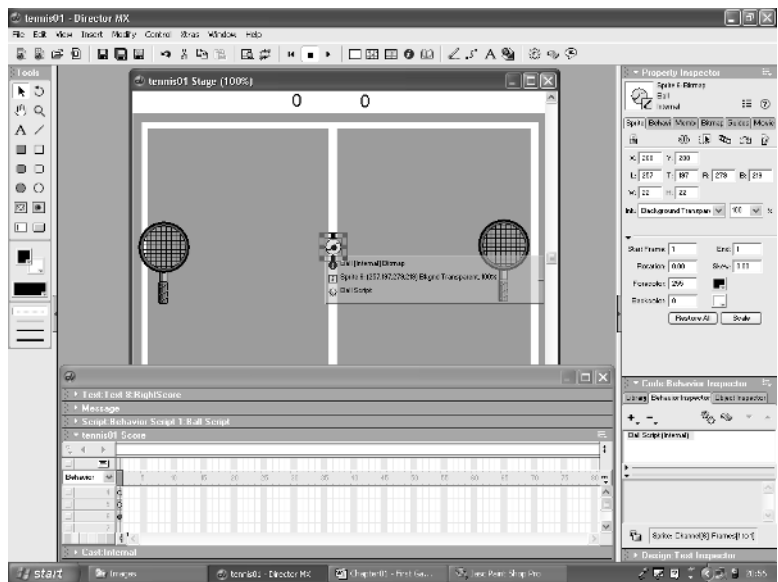


Figure 1.1 The Director interface



Figure 1.2 The Control Panel window

You can alter the speed of the movie by changing the value of ‘fps’ (frames per second). A higher value gives a faster movie.

The first thing to do when following this tutorial is to open the file ‘Examples/Chapter01/tennis01.dir’ from the CD. You must save the project to your local machine if you intend to edit and keep a project. You should be looking at something similar to the view shown in Figure 1.1. The first step in creating the game will be to draw the racket, ball and background. To do this we will open the ‘Paint’ window. Select ‘Window/Paint’ from the main menu bar or press ‘Ctrl + 5’. Figure 1.3 shows the Paint window in Director MX 2004, with the last of the assets we are going to create, the racket.

Creating the ball

The first stage of this tutorial chapter is creating the ball; the movie ‘Examples/Tennis01.dir’ already contains the court. When creating a game in Director we first create the graphic assets, which are stored in the ‘Cast’. You can view the contents of the Cast at any time by opening the Cast window. Select ‘Window/Cast’ from the main menu bar or press ‘Ctrl + 3’. Figure 1.4 shows the Cast window from the final version of the tennis game.

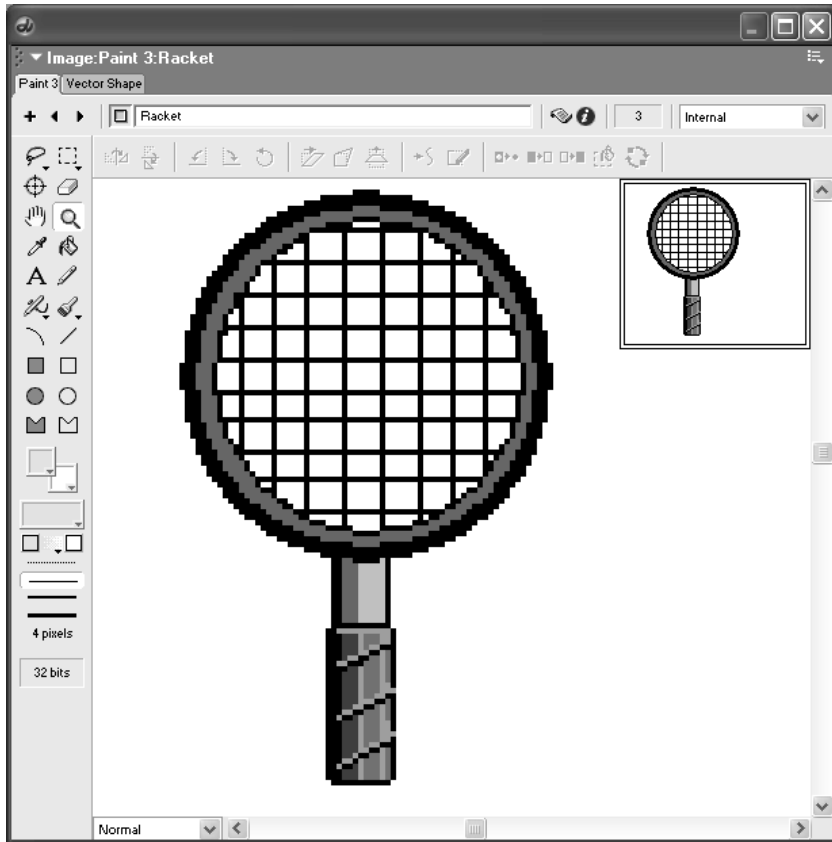


Figure 1.3 The 'Paint' window in Director MX 2004

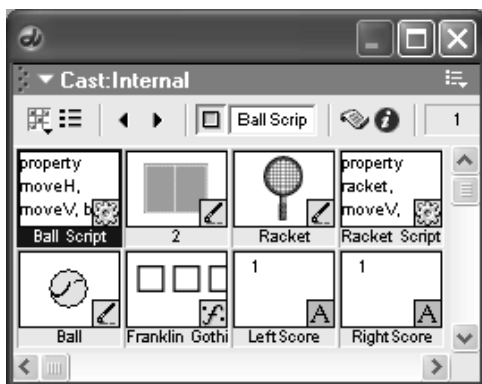


Figure 1.4 The 'Cast' window in Director MX 2004

OK, you're sitting in front of your computer, you've tried opening a few different windows from the 'Window' drop-down menu on the main menu bar and now you are ready to start work. Move to the 'Paint' window and press the '+' button (see Figure 1.5).

This adds a new bitmap for you to work on. As you add new bitmaps to the 'Cast', you can move through these using the arrows (see Figure 1.6).

You can give the bitmap a useful name; for this tutorial call it 'Ball'. Enter the name in the text box highlighted with a circle in Figure 1.7. When there are only a few bitmaps and you are just creating a game by yourself you may be tempted to take the default name. Fight this!



Figure 1.5 Adding a new bitmap using the 'Paint' window



Figure 1.6 Moving through bitmaps using the arrow buttons



Figure 1.7 Naming the bitmaps using the 'Paint' window

Get into the habit of naming everything; if the project gets bigger then you will be glad you did. Also you may come back to a movie months after it was completed and it will be much easier to find your way around if everything has a useful name. If you are working in a team then you will enjoy working on someone else's project if the project is well organized with easily identifiable names; if everything is poorly structured then the project becomes a nightmare to maintain.

Before we do any drawing in Director, let's briefly look at each of the tools in the 'Tools' palette (see Figure 1.8). Starting in the top left, the 'Lasso' allows you to select an irregularly shaped area of the bitmap. The 'Marquee' lets you select a rectangular area that can then be modified using various stretch subtools found in the bar at the top of the 'Paint' window. The next row down has the 'Registration Point' tool; this affects the alignment of a bitmap. The 'Eraser' lets you rub out any mistakes. Moving down you will find the 'Hand' tool, which lets you move the artwork around and the 'Magnifying Glass', which allows you to zoom into the artwork for fine tuning. Next down you will find the 'Eyedropper' to select colors from the bitmap and the 'Paint Bucket' to flood closed areas of a bitmap with the selected color. The 'Text' tool allows you to enter text into the bitmap and the 'Pencil' lets you draw. The 'Air Brush' creates a splatter cap spray and the 'Brush' allows you to draw with one of five user-definable brushes. 'Arc' and 'Line' create curved or straight lines. 'Filled Rectangle' and 'Rectangle' create rectangles. The 'Ellipse' and 'Polygon' tools create outline or filled ellipses and polygons. The 'Foreground Color' and 'Background Color' buttons let you define the colors that you use to draw with and 'Gradient Colors' defines a more complex fill option. Below this you can define the line style and set the 'Color Depth' of the bitmap.

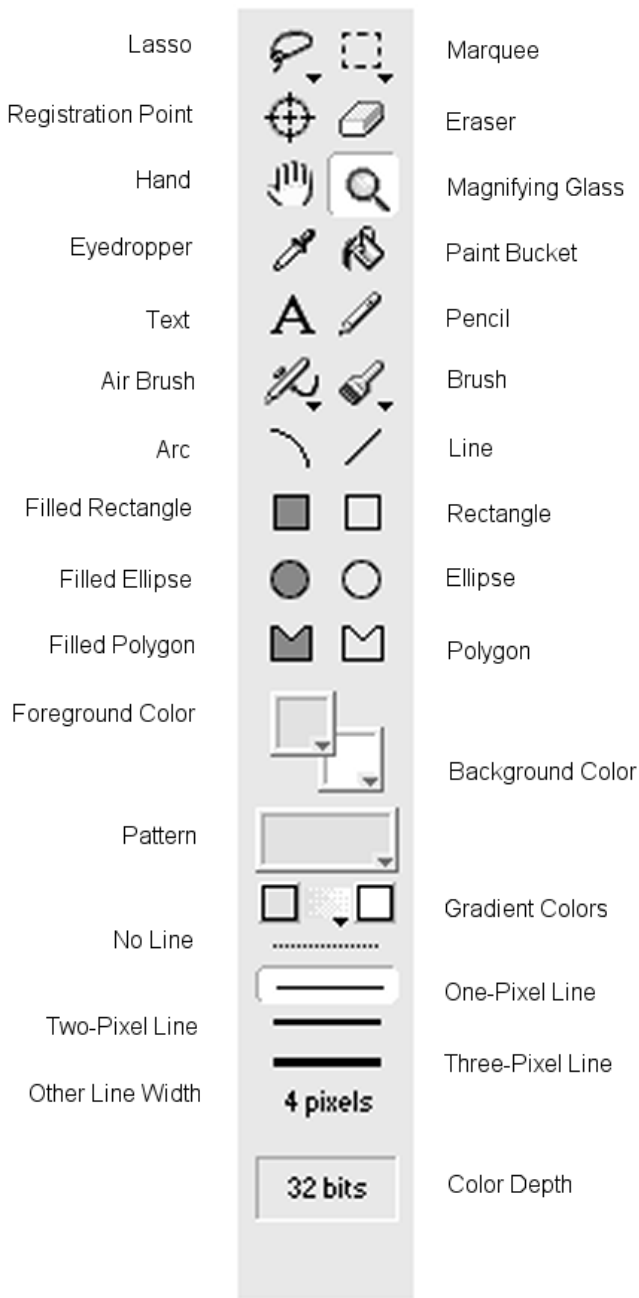


Figure 1.8 The ‘Tools’ palette used in the ‘Paint’ window



Figure 1.9 *Selecting the 'Filled Ellipse' tool*



Figure 1.10 *Selecting colors*

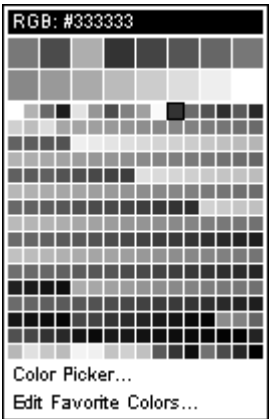


Figure 1.11 *Choosing colors*

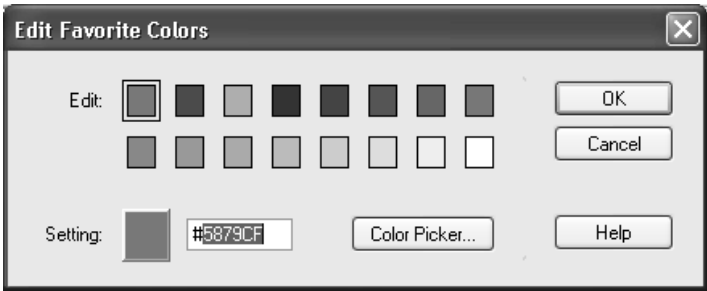


Figure 1.12 *Editing your favorite colors*

Some of you may well be familiar with paint programs. If so then you may find the 'Paint' window a little disappointing. We are going to use it for this introductory chapter, but from then on we will be using Director as the place to bring all our assets together. To create these assets we will use several other programs. You will find that Adobe Photoshop, Paint Shop Pro or other dedicated paint programs provide much more power when creating the artwork for your games leaving Director to do just the occasional retouch.

With the 'Filled Ellipse' tool selected (see Figure 1.9), choose a 'Foreground Color' for the ball (see Figure 1.10).

Select the top left square. The little arrow in the bottom left of this button indicates that clicking it will bring up further options. In this case it opens the basic color chooser (see Figure 1.11).

The top 16 colors are your favorite colors selected, which can be edited using the 'Edit Favorite Colors' option at the base of the window (see Figure 1.12). A menu option with a trailing '...' again indicates that a new dialog box will open when clicked.

The favorite color edit dialog contains buttons for the 16 colors, a button to open the color picker, which you will learn more about later and a place to enter a hex value. Web developers will be familiar with specifying colors using hexadecimal. Once all your favorite colors have been selected click 'OK' to continue, or 'Cancel' to abort the edit without saving the results.

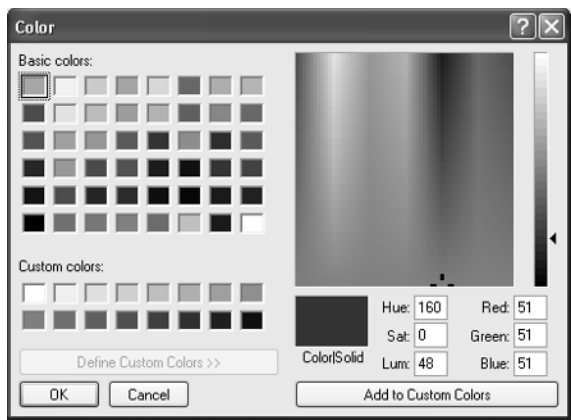


Figure 1.13 The ‘Color Picker’ dialog box

You can select the ‘Color Picker’ directly from the basic color palette or from the ‘Edit Favorite Colors’ dialog. Either way the dialog box in Figure 1.13 opens. This is a system-level dialog box and so varies between a Windows and a Mac platform; all illustrations in this book show the Windows dialog boxes.

Move the arrow in the large color display to set the basic color. The saturation of the color is maximum at the top and minimum at the bottom. Then slide the luminosity control at the right. You will see the values of ‘Red’, ‘Blue’ and ‘Green’ change as you do. A common way to specify colors is by selecting a Red, Green and Blue level. Each of these varies between 0 and 255, a strange choice you may think, but in actual fact this is the range that can be stored in a single byte.

A byte contains eight bits. Each of these bits can contain either 0 or 1. Computers use binary to store everything from images to documents to music. If we combine two binary bits then we can combine these so that one represents 0 or 2 and the other 0 or 1. Here is a table of all possible combinations of these two bits.

2	1	Total
0	0	0
0	1	1
1	0	2
1	1	3

Just as in the number 100 the position of the 1 at column three makes it have the value $10 \times 10 \times 10$, in binary the position of the bit defines its value. By using two bits we can specify numbers 0, 1, 2 and 3. If we pack eight bits together and set the bits of each we get the number 255, which is why ranges of values between 0 and 255 are common.

128	64	32	16	8	4	2	1	Total
1	1	1	1	1	1	1	1	255

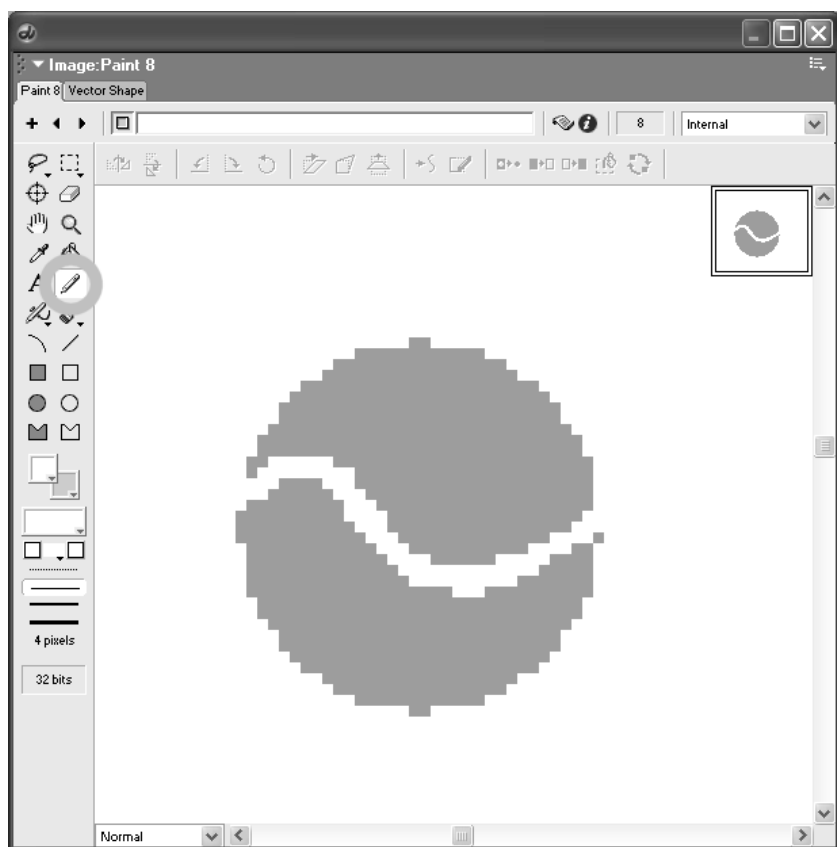


Figure 1.14 Using the 'Pencil' tool

Choose the color you want for the ball and then click to define the top left of an imaginary rectangle that surrounds the circle; now keeping the mouse button down drag to define the size of the circle. You will see the circle being drawn.

Choose the 'Pencil' tool and draw a line through the ball as shown in Figure 1.14. The graphic artists amongst you will be troubled by the lack of anti-aliasing. You can see stepping in the image. In the screen grab you can see two sizes of ball, the smaller one showing the actual size the ball is to be used. By looking at the enlarged version you can see how the image is made up. Even the highest resolution screen is still poor by print standards and to give the illusion of a smoother image the edges can be blended as shown in Figure 1.15. This ball was created using Paint Shop Pro which has the option with most tools to use anti-aliasing. You can see that instead of the edge being stepped it is made up of ranges of colors, which when viewed at the screen resolution blend to give the impression of a smooth curve.

Unfortunately this facility is lacking from the 'Paint' window in Director MX 2004. For this reason and because of several other failings, it is not recommended as a full production tool for professional work; you are advised to use an alternative program and then import the artwork, as was done to provide the image for Figure 1.15. We will look at importing images later in the book.

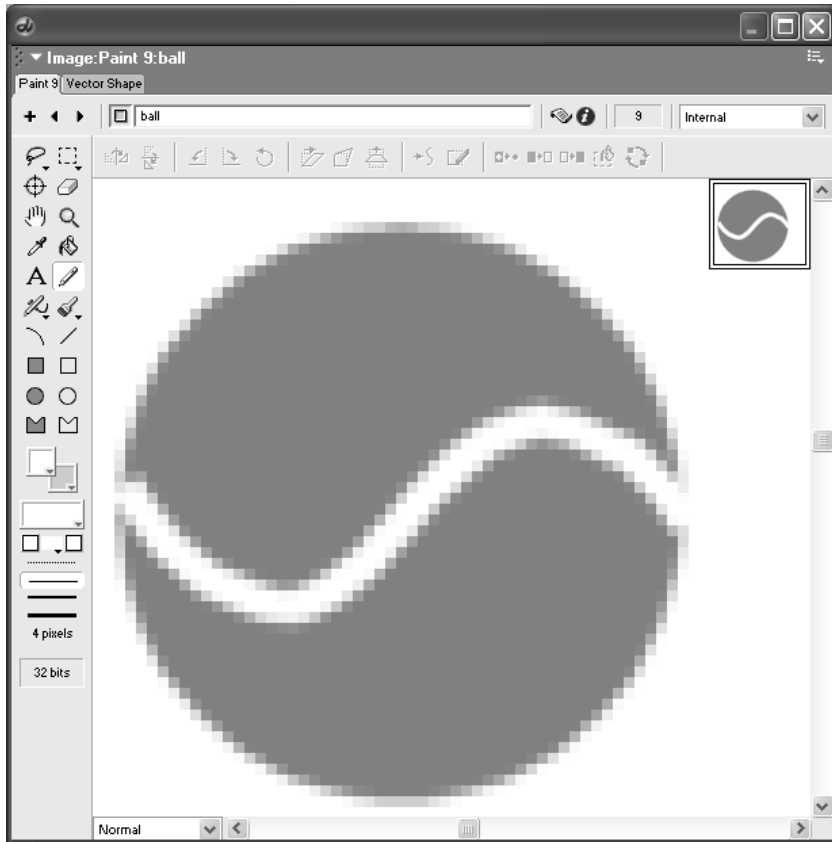


Figure 1.15 *Anti-aliasing to improve the smoothness of an image*

Adding the rackets

At this stage you have options; you can try to create the racket as shown in Figure 1.16 or you can take the easy way out and open the project from the file 'Examples/Chapter01/Tennis02.dir' on the CD.

Using text

Director has many tools available for using text, but before we use them we want to ensure that the font we select is available on the user's system. If it is not available then they will get a replacement font that is likely to be very different from the fun font you have carefully chosen. We need to embed a copy of the font so that the published game contains the font outlines that are needed for the game.

Select 'Insert/Media Element/Font...' from the main menu bar to open the dialog box shown in Figure 1.17. Select the font you are embedding using the drop list 'Original Font:'. You can select which style to embed: 'Plain', 'Bold', 'Italic' or 'Bold Italic'. If you want to use bitmap fonts then you can choose a size. Finally you can select which outlines to embed, either the entire set or

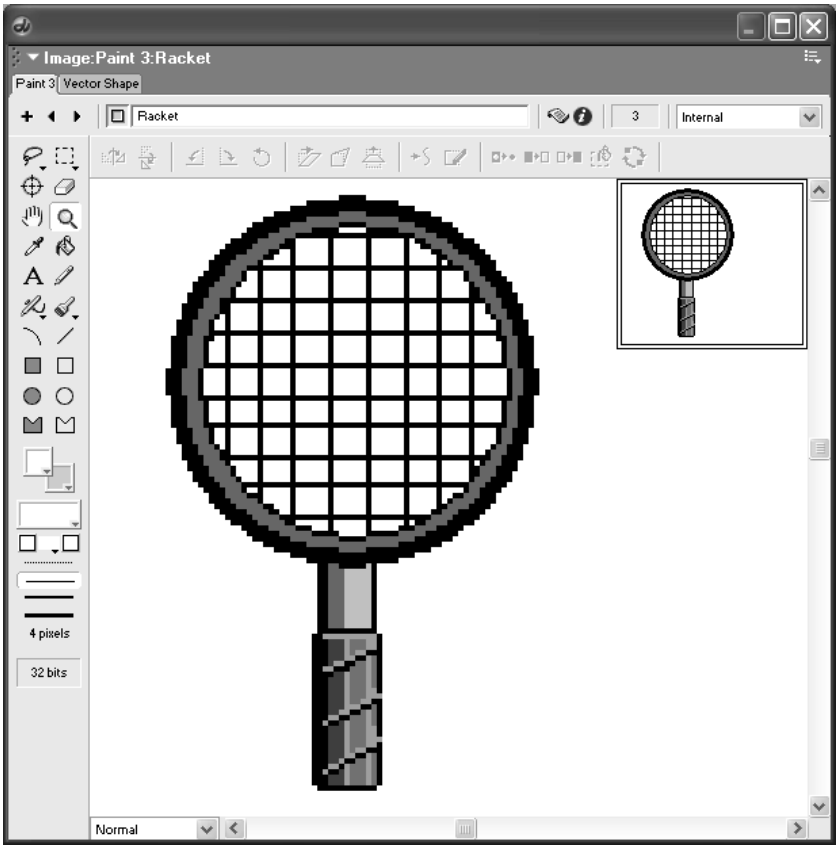


Figure 1.16 The racket

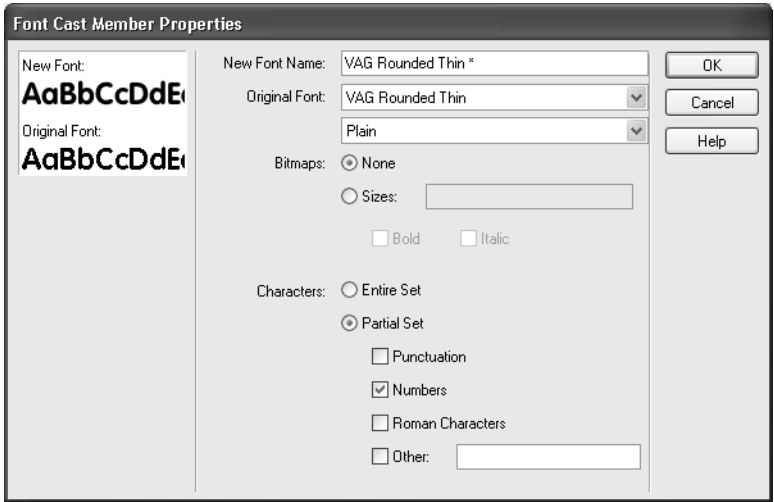


Figure 1.17 Embedding a font into the 'Cast'

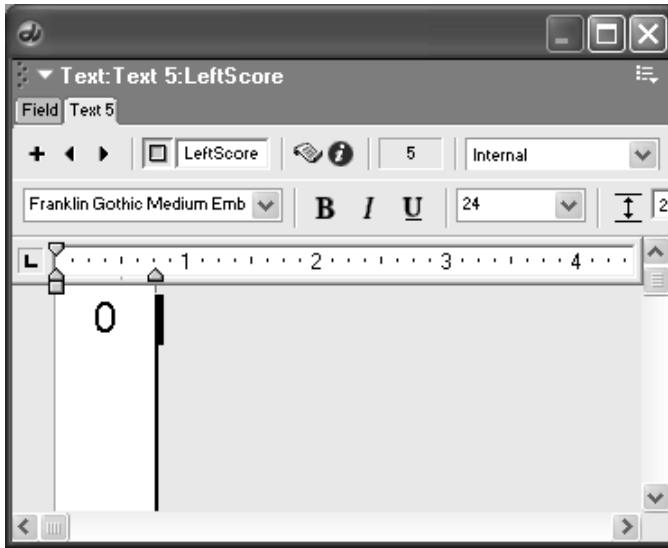


Figure 1.18 Using the ‘Text’ window

a limited selection. For the current game we only need numbers. Limiting the selection makes the game that bit smaller, which may be important for a web-based game, particularly one that uses several different fonts. A typical full font set adds around 20 K to the game. If you only ever use the letters ‘GAME OVER’ from that font then simply put ‘GAMEOVR’ in the ‘Other’ box to limit the size by a worthwhile amount. By default, embedded fonts take the original font name and add an asterisk (*). But you can give the font any name you choose. I tend to add ‘embedded’ to the name to remind myself. When you have chosen a font press ‘OK’. If you now examine the ‘Cast’ window you will see a font cast member; the icon indicates the type of cast member and a small ‘f’ indicates a font.

To create a text cast member, open the ‘Text’ window (see Figure 1.18) by selecting ‘Window/Text’ from the main menu bar or by pressing ‘Ctrl + 6’. Press the ‘+’ button to create a text cast member and name it ‘LeftScore’. Select the embedded font from the drop-down list. Make sure you select the embedded font not the original; they will both be in the list. Now set the point size to 24 and type ‘0’ in the text area. If you are unhappy with the color then open the ‘Tools’ palette window (see Figure 1.19) by selecting ‘Window/Tool Palette’ or by pressing ‘Ctrl + 7’.



Figure 1.19 Using the ‘Tools’ palette to set the text color

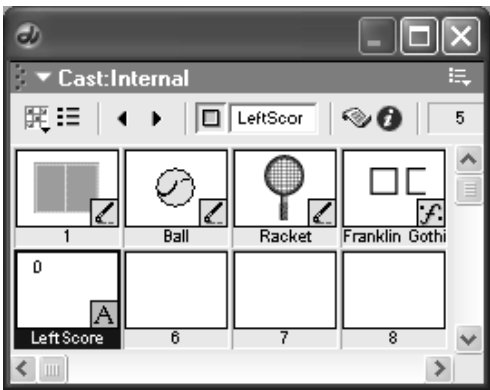


Figure 1.20 Selecting a cast member

The color selection uses the same mechanism as discussed for drawing the ball.

To create a text cast member for the other player, click on the cast member 'LeftScore' in the 'Cast' window (see Figure 1.20). Press 'Ctrl + C' for copy, then click on an empty cast place and press 'Ctrl + V' to paste. You could use the menu options 'Edit/Copy Cast Member' and 'Edit/Paste' to do the same thing. Rename the new cast member as 'RightScore' by typing in the name box in the 'Cast' window. At this stage if you have been working through from version 01 in the CD project files then you will only have the court on the 'Stage'. To move elements from the 'Cast' window to the Stage simply drag and drop them. Start with the ball, placing this in the middle of the Stage window. Open the 'Property Inspector' window by selecting 'Window/Property Inspector' from the main menu bar. If you have the default setup then dragging the ball onto the Stage will place it there for 30 frames. You can change the beginning and end of this sprite span using the Property Inspector window (see Figure 1.21).

Enter 1 for this tutorial. Notice that the ball shows white corners around its edges. To avoid this we must set the background color to be transparent (see Figure 1.22).

Use the 'Ink' drop-down and select 'Background Transparent'; the background now shows through around the outside of the ball. Repeat this procedure

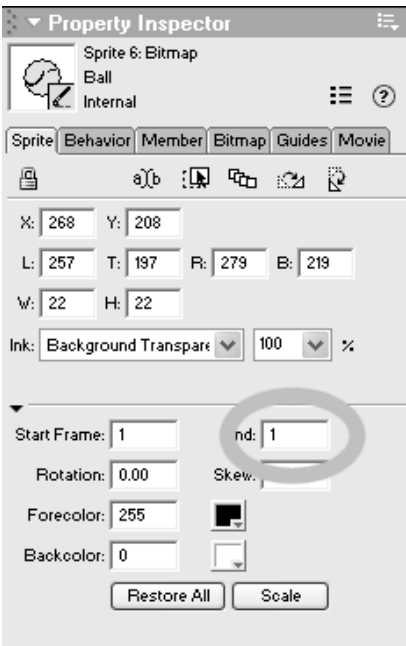


Figure 1.21 Changing the end frame using the 'Property Inspector' window

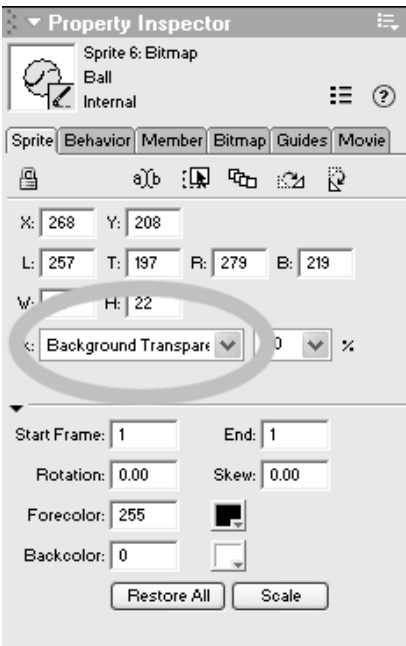


Figure 1.22 Selecting the ink using the 'Property Inspector' window

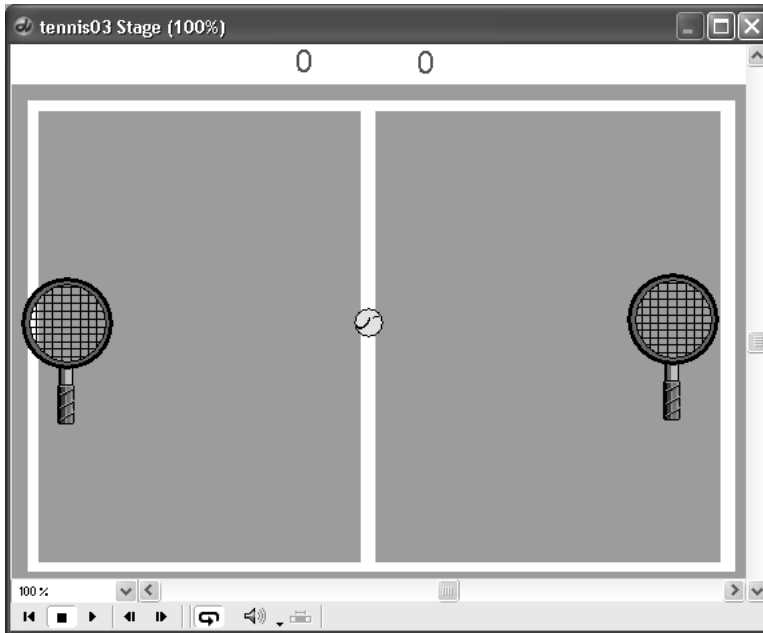


Figure 1.23 *All the elements in place for the Tennis game*

for both of the rackets. Finally add the text cast members ‘LeftScore’ and ‘RightScore’ until the ‘Stage’ window looks like Figure 1.23.

We are now ready to create the game.

Adding a little code

Each element on the ‘Stage’ is in a separate channel. Take a look at the ‘Score’ window (see Figure 1.24) and you will see that there are several rows occupied. Try selecting one of these; you will see that the element is selected on the Stage.

Each row in the ‘Score’ window is a sprite channel. Selecting sprite channel 4 in Figure 1.23 highlights the left racket. We now need to add a ‘Behavior’ for this sprite. To do this, click the Behavior tab in the ‘Property Inspector’ window (see Figure 1.25).

Pressing the ‘+’ button opens a menu. At this stage the menu contains just the option ‘New Behavior...’, which opens a small dialog box where you can name the Behavior. Call it ‘Racket

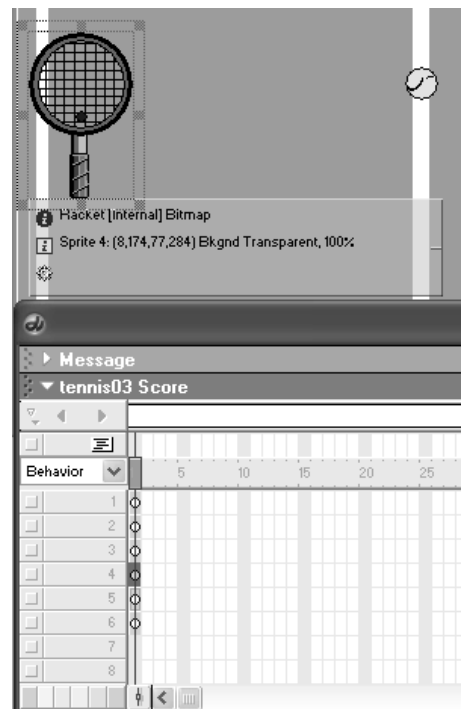


Figure 1.24 *Selecting using the ‘Score’ window*

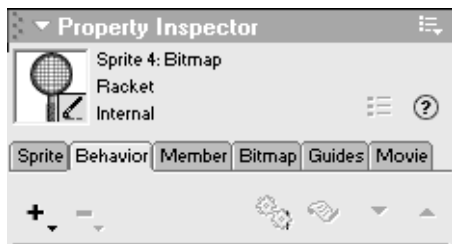


Figure 1.25 Using the 'Property Inspector' to add a new 'Behavior'

the Behavior tab of the Property Inspector window, select New Behavior... and call it 'Ball Behavior'. If you now look at the 'Cast' window you will find that there are now two new cast members, Racket Behavior and Ball Behavior. Unfortunately these Behaviors have no idea what is required of them; they just represent a place where we will add the code necessary to move the rackets and the ball.

Now is the time to start to enter a little Lingo, the scripting language that Director MX 2004 uses to control the sprites. Right click on the cast member 'Racket Behavior' and select 'Cast Member Script...' from the context menu. Alternatively select 'Window/Script' from the main menu bar and use the arrow buttons to move through the scripts in your 'Cast' to find the one called 'Racket Behavior'. At this stage the 'Script' window is empty awaiting your code. To enter code, simply type it directly into the Script window. Here is the first bit of code to enter:

```
1 property racket
2
3 on beginSprite me
4   racket = sprite(me.spriteNum)
5 end
```

Listing 1.1

Do not enter the line numbers at the beginning of each line; they are simply there so that a line can be identified when the explanation is described in the text. The first key word we find is 'property'; any word that follows that will become a variable, a storage spot for a number, a word or sentence or an object. In this instance we use the name 'racket'. We are going to use this to refer to the sprite that contains the racket image. Having declared the variable it does not yet refer to anything. We need to set its value. For this we use the handler 'beginSprite'. Director defines events and functions using the syntax

```
on handlerName
  --Enter the Lingo code
end
```

You can choose any name you wish for 'handlerName', but certain events are predefined; beginSprite is one such event. If this handler is defined in a behavior then the code between the

event name and 'end' will execute as soon as the sprite is found on the 'Stage'. The `beginSprite` event, in common with all events, is passed the parameter 'me'; me has a number of useful properties, the one used in Listing 1.1 at line 4 is 'spriteNum', which returns the channel number for the sprite using this behavior. Lingo stores list of all sprites on the score, so we can set the 'property' racket to the actual sprite by assigning it the value 'sprite(me.spriteNum)', the sprite with the channel number that matches the user of this behavior. Now that the variable racket is set we can use it to refer to the sprite.

Another important event is the 'enterFrame' event, which occurs for every frame in the movie. If you have set your movie to play at 30 fps then this event will occur 30 times in a second ideally; on computers that cannot maintain this speed then the fastest speed that can be attained will be used. We are going to use the `enterFrame` event to test for key presses and move the racket if certain keys are pressed. Lingo contains the method 'keyPressed'; this takes a parameter, which is always put in parentheses. If there is more than one parameter, they are separated by commas. The method `keyPressed` takes a single parameter, the key, usually indicated by the letter between quotes. If you want to test for the pressing of the key 'K', you would use the syntax

```
keyPressed ("K")
```

This method returns a value 'true' if the key is pressed and 'false' if it is not. We can use this information to move the racket. We are going to use the keys 'A' to move the racket up the screen and 'Z' to move the racket down the screen. So we need a condition that happens when the `keyPressed` function returns true. We can use the Lingo 'if ... then' structure, one of the most useful of all programming structures. The if ... then structure uses the following syntax:

```
if (condition) then
  --Do something
else
  --Do something else
end if
```

where 'condition' can be anything that evaluates to 'true' or 'false'. By placing '`_key.keyPressed("A")`' where 'test' is shown we now have the syntax necessary to test for the A key:

```
1 property racket
2
3 on beginSprite me
4   racket = sprite(me.spriteNum)
5 end
6
7 on enterFrame
8   if (_key.keyPressed("A")) then
9     --Move racket up
12  end if
13 end
```

Listing 1.2

If the A key is not pressed then we must test for the Z key; we can extend Listing 1.2 to do this:

```

1 property racket
2
3 on beginSprite me
4   racket = sprite(me.spriteNum)
5 end
6
7 on enterFrame
8   if (_key.keyPressed("A")) then
9     --Move racket up
10  else if (_key.keyPressed("Z")) then
11    --Move racket down
12  end if
13 end

```

Listing 1.3

Notice line 9 in Listing 1.3. By preceding any words in a line with ‘--’ they become a comment; Lingo ignores the remainder of the line, but you can read the words. Commenting your code can make maintenance of the code much more convenient. If you return to a project or worse still take over someone else’s project then you will appreciate richly commented code. Otherwise you will have to work out what the code does. This can be both irritating and time consuming. Now we need to actually move the racket. Every sprite has many properties. A property is accessed using dot syntax, which means you give the variable name for the sprite, followed by a dot, then the property name. In this simple game the rackets can only move up and down so we simply need to adjust the value for the vertical component. This is called ‘locV’, location vertical. Sprite locations increase down the screen. So to move the racket up the screen we need to decrease the value of locV and to move the racket down the screen we must increase the value of locV. The syntax we use is

```
racket.locV = racket.locV - 1
```

This will move the racket up the screen by 1 pixel.

```

1 property racket
2
3 on beginSprite me
4   racket = sprite(me.spriteNum)
5 end
6
7 on enterFrame
8   if (_key.keyPressed("A")) then
9     racket.locV = racket.locV - 1
10  else if (_key.keyPressed("Z")) then
11    racket.locV = racket.locV + 1

```

```

12 end if
13 end

```

Listing 1.4

Now you can finally test your handiwork. Press the ‘Play’ button on the main toolbar or select ‘Control/Play’ from the main menu bar. Now if you press the A or the Z key then the racket moves up or down the screen. However you might notice a problem. The left racket and the right racket share this behavior so they both move at the same time. It would be better if the left racket used one pair of keys and the right racket an alternative choice. How do we make this happen?

First we will define two new properties for the behavior, ‘upKey’ and ‘downKey’. We will change the ‘beginSprite’ event to assign different keys based on the channel for the sprite using this behavior. The left racket is in channel 4, so we can test for this and assign the keys ‘A’ and ‘Z’ if the sprite is in channel 4. If it is not then we will use the keys ‘K’ and ‘M’.

```

1 property racket, upKey, downKey
2
3 on beginSprite me
4   racket = sprite(me.spriteNum)
5   if (me.spriteNum = 4) then
6     upKey = "A"
7     downKey = "Z"
8   else
9     upKey = "K"
10    downKey = "M"
11  end if
12 end
13
14 on enterFrame
15   if (_key.keyPressed(upKey)) then
16     racket.locV = racket.locV - 1
17   else if (_key.keyPressed(downKey)) then
18     racket.locV = racket.locV + 1
19   end if
20 end

```

Listing 1.5

If you run the program after the changes made in Listing 1.5 then you will see that each racket responds to its own keys. Unfortunately the racket’s movement is a little slow. We want it to speed up if we keep pressing a key. This can be achieved by using a custom property, ‘moveV’. This is initialized to zero in the beginSprite event and then used to control the speed in the ‘enterFrame’ event. We also need to test for the racket going out of screen at the top or bottom. At the top we will stop the racket if its ‘locV’ property is less than 20. The symbol ‘<’ is used to test for less than. Line 17 in Listing 1.6 shows how this is done, the moveV property is set to zero if the condition ‘racket.locV < 20’ (the vertical location of the racket is less than 20) is met. If it is not, then we

check to see if the moveV property is greater than -15 . The symbol ' $>$ ' is used to test for greater than. Line 19 in Listing 1.6 shows how this is done. This uses the syntax 'else if', that is, a previous 'if' test has failed so we now have a second test. If moveV is greater than -15 , we decrease the value of moveV by one. The same principle is used for moving up the screen. If neither the upKey nor the downKey are pressed then we slow the racket down by multiplying the value of moveV by 0.8, which has the effect of making it smaller. Once moveV is less than 1 we set it to zero at line 30. Now that moveV is used for moving we adjust the location of the racket using line 33.

```

1  property racket, moveV, upKey, downKey
2
3  on beginSprite me
4    racket = sprite(me.spriteNum)
5    moveV = 0
6    if (me.spriteNum = 4) then
7      upKey = "A"
8      downKey = "Z"
9    else
10     upKey = "K"
11     downKey = "M"
12   end if
13 end
14
15 on enterFrame
16   if (_key.keyPressed(upKey)) then
17     if (racket.locV < 20) then
18       moveV = 0
19     else if (moveV > -15) then
20       moveV = moveV - 1
21     end if
22   else if (_key.keyPressed(downKey)) then
23     if (racket.locV > 380) then
24       moveV = 0
25     else if (moveV < 15) then
26       moveV = moveV + 1
27     end if
28   else
29     moveV = moveV * 0.8
30     if (moveV < 1) then moveV = 0
31   end if
32
33   if (moveV < > 0) then racket.locV = racket.locV + moveV
34 end

```

Listing 1.6

This completes the code for the racket. The code for the ball is more complicated and is best left until you have more Lingo under your belt. But take a look at 'Examples/Tennis05.dir' to see how the final game works and check out the code for the ball. Notice that the score for the opposite player is incremented as the other side misses a shot.

How to improve it

To finish the game there needs to be a beginning and an end. The beginning would give instructions for key presses and the end would show the final score. In this simple example we have excluded these details but you are encouraged to develop the game further to include these aspects as your skills progress.

Playing against the computer

It would be nice if the player could compete with the computer. To achieve this one of the rackets could be under computer control; the behavior for the racket would judge where the ball would hit the racket and move the racket accordingly. Although this is a little complicated it should be within your capabilities after completing Section 3 of this book. When the computer is in control it may be impossible for the player to win. It is your job as a developer to set the skill level so that a player is not frustrated, but is challenged. Setting the game play level so that a player is progressively challenged as their skill level develops is one of the hardest aspects of game play development, but one of the most important.

Summary

This was a quick look at creating a Director game; hopefully it gave you an overview. If you found the code confusing, don't worry, we will be going at a much slower pace in the later chapters when looking at Lingo in detail. In principle I want you to take away from this chapter an initial awareness of the Director interface, how to navigate the many windows that Director uses, the use of sprites and where to enter your Lingo. As you progress through the book you will be encouraged to enter your own code and develop the examples that you will find in later chapters. Nothing improves your skill more than trying it out yourself. Practice really does make, if not perfect, then at least much better.

This page intentionally left blank

Section 1

Layout

The first section looks at creating the artwork and animation that will feature in the games we create.

This page intentionally left blank

2 Images and computers

When creating a game you need a combination of skills: art, sound and code. The only one of these skills where you will use Director as an application is the last one, programming. But before you can add a line of code you will need graphics, animation and in most cases, audio. As we learned in Chapter 1, Director is mainly a tool for bringing together these assets rather than creating them. You are going to need to use several different programs along with Director to create any worthwhile game. If you are creating a 2D game then you will need to learn to use a paint program and a sound editor at the very least. If you are creating a 3D program then you will also need to learn how to use a 3D modeling and animation package. In this chapter we are going to discover the options available when creating sprite graphics, the irregularly shaped images that will form the basis for all 2D games whether they are board games, quizzes or arcade-style games. This book is principally a technical guide, so we are not going to try to show how to use a specific art package or guide your design skills; you can find many books that will offer advice for using Adobe Photoshop, for instance. Instead we are going to look at general approaches and examine the many options available when creating art on a computer. Before we get too deep we will need to look at file formats to show how one approach can be used for some solutions and another will work in an alternative situation.

Bitmap or vector

The first lesson you need to learn is that Director is principally a bitmap package. Take any digital camera that you can buy at your local electronics store and you will see that it boasts a certain number of pixels. Look at the ads promoting the cameras and the term megapixel is never far away. A megapixel describes a million pixels. If the resolution of a camera is 1000 across and 1000 down, then the total resolution is 1000×1000 , or one million pixels. A pixel is simply a square of color. Take a look at Figure 2.1, which shows a tiny detail of the photograph in Figure 2.2.

Each pixel or square of color is given a value for red, green and blue. In many instances, for reasons of computer history, the value of a single channel of color is a number between 0 and 255. To understand why this is, we need to learn two useful computer terms: *bit*, the smallest unit of

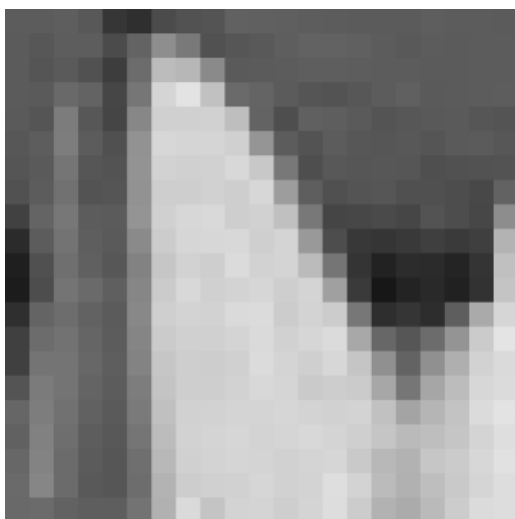


Figure 2.1 *Pixels in close-up*

data that can be stored on a digital computer, either the number 0 or 1, and *byte*, a collection of eight bits that can represent any whole number (integer) between 0 and 255.

So how is a byte used to represent all those numbers? Basically a computer is a hugely elaborate switch. Just as a switch can either be on or off, computers work as a bank of switches. Link eight switches together and you can assign a value to each switch. The first switch represents either 0 if it is off or 1 if it is on, the second switch can be used to indicate 2 if it is on and 0 if it is off. The next switch indicates 4 or 0 and the next 8 or 0, followed by 16 or 0, 32 or 0, 64 or 0 and finally 128 or 0. Each switch shows double the previous if it is active. If we turn switches 0 and 2 on then we will get 1 (switch 0 in the on position) and 4 (switch 2 in the on position). This equals $4 + 1 = 5$. By combining all the switches 0, 1, 2, 3, 4, 5, 6 and 7 we are summing $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$. A collection of eight bits, also called a byte, can store the numbers 0 to 255.

As you can see from Table 2.1,

- 128 + 1 = 129 or the binary number 10000001
- 2 + 1 = 3 or the binary number 00000011
- 0 = 0 or the binary number 00000000
- 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255 or the binary number 11111111

Using just 256 different values for a color might not sound like a lot, until you realize that this gives 256 alternative values for red, 256 for green and 256 for blue. The total number of alternative

Table 2.1 Using binary numbers

128	64	32	16	8	4	2	1	Total
1	0	0	0	0	0	0	1	129
0	0	0	0	0	0	1	1	3
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	255



Figure 2.2 The photograph shown enlarged in Figure 2.1

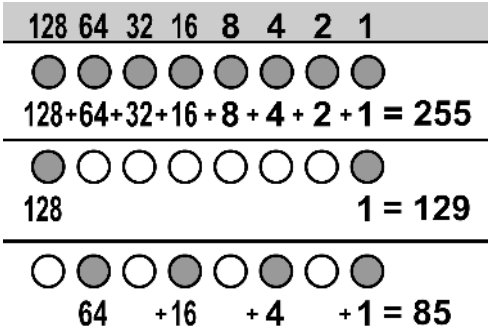


Figure 2.3 How numbers are stored in an image file

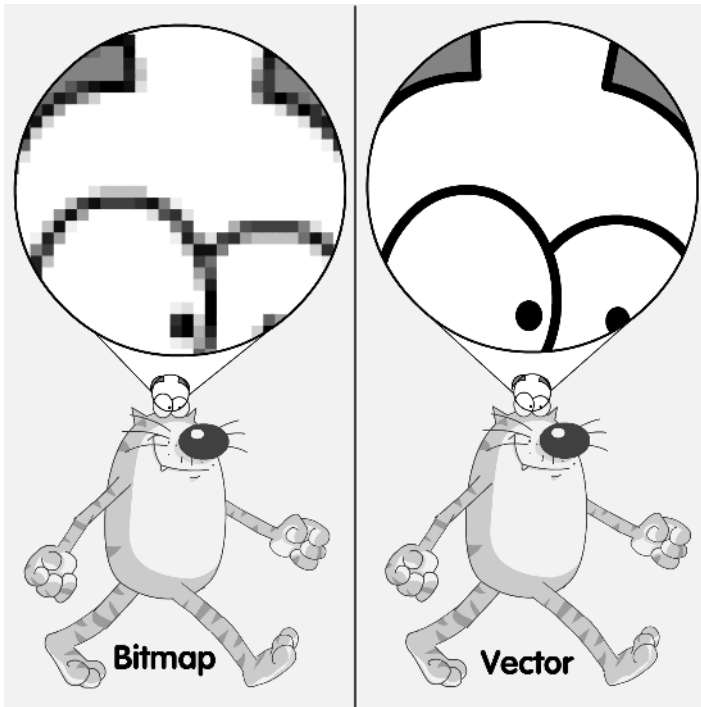


Figure 2.4 *Comparison of bitmap and vector graphics*

colors is therefore $256 \times 256 \times 256 = 16\,777\,216$, nearly 17 million different colors, which sounds an awful lot more than 256.

A bitmap is just one way to store graphics information and is ideally suited to a photographic image where each pixel in a line may be different. But a nonphotographic image can also be stored either as a bitmap or a vector image.

Figure 2.4 shows the difference between a bitmap and a vector image. A bitmap is simply a grid of squares, each one of which is given a color value. A vector image is a set of instructions that the computer uses to rebuild the image. Looking at the illustration you wonder if there can ever be a benefit to a bitmap image. The more we look closely at the vector image the smoother the result, a bitmap image just breaks down into a single dot. That is very definitely true. But look also at the overall image of the strange walking cat that we will see more of in the book; when seen in full there is no difference between the bitmap version and the vector version. Before we can see how this is going to be a benefit we need to consider a computer display.

How is the image shown on the computer?

A computer display is simply a grid of those popular units of color, the pixel. The resolution of your display can be 640 pixels across and 480 down (640×480) or 800×600 , 1024×768 , and so on. As you move to more pixels in the display so the icons get smaller and the fonts can get more difficult to read. Figures 2.5 and 2.6 show how the same panel appears smaller simply by changing the resolution.

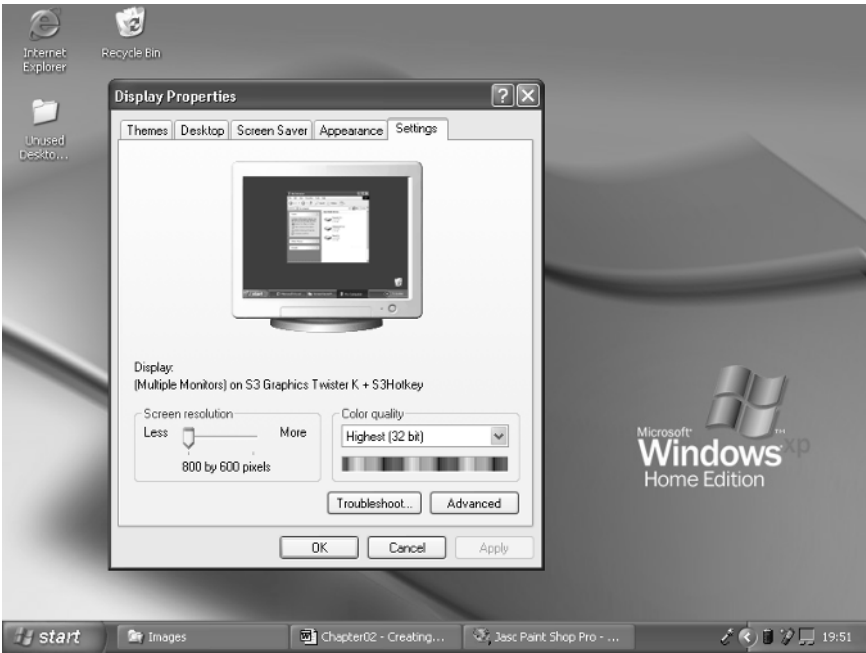


Figure 2.5 Desktop at 800 × 600

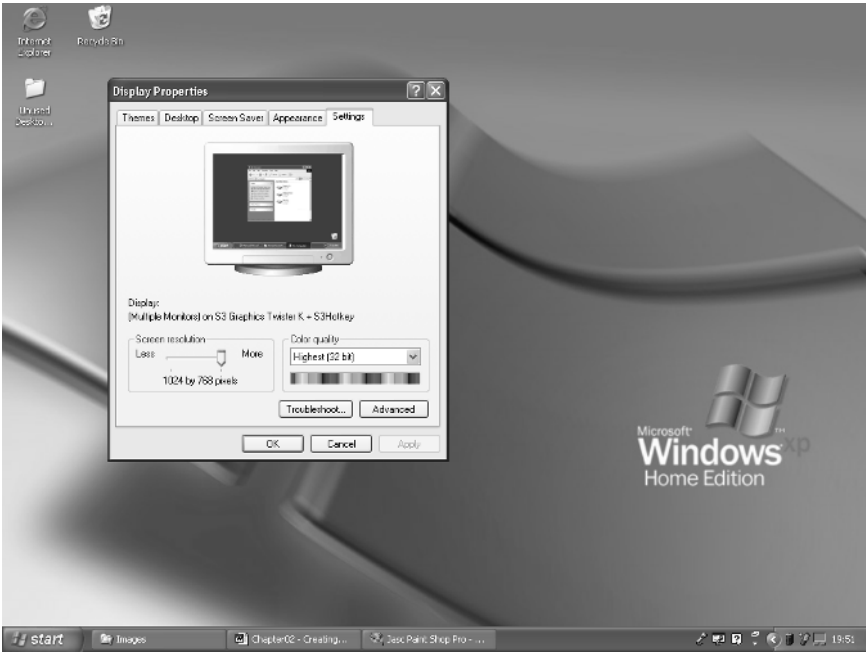


Figure 2.6 Desktop at 1024 × 768

When creating games you are usually targeting a specific resolution. If we make a game where the maximum resolution across is 640 then a single character graphic, a sprite, is unlikely to be more than 150 pixels wide. Let's now consider the cat in Figure 2.4. If this image is 150 pixels wide and 150 pixels high then the computer needs to draw $150 \times 150 = 22\,500$ pixels. This might sound like a lot but graphics cards are great at drawing pixels. The vector alternative needs to draw 878 curves, so you might think this sounds like the vector one will be easier. But a vector option needs to use the 878 mathematical curves in order to create a grid of pixels, whereas with the bitmap option we already know what the pixels are. If you do not intend to scale an image and the detail in it is quite high then a bitmap will always be faster. If scaling is to be used extensively in a 2D game then Director might not be the best option. If scaling is not used and fast redraw times are important then Director is a good choice without getting your hands too dirty using C++ to write your games.

So what are all those different formats?

OK, we are going to create a nonscaling fast action game using bitmap images, so what are the differences between jpeg, tif, tga, gif, bmp, png, eps and ai files? Before we go any further we need to look more carefully at how color is stored in an image. If we use eight bits of information for the red, green and blue in an image then we have an image with eight bits for the red, eight bits for the green and eight bits for the blue. This gives us a total of 24 bits. You have probably heard of 24-bit color and this is where the term originates. The simplest possible image file could start by saying how many pixels across and down and then just have a list of each pixel:

```
--Header
16 - width
16 - height
--Pixel data
0, 0, 0 - first pixel on the first row
255, 255, 255 - second pixel on the first row
...
```

This technique is essentially how an uncompressed bmp file is stored. Another possibility is to use a color table:

```
--Header
16 - width
16 - height
--Color Table
0, 0, 0 - Color index 0
255, 255, 255 - Color index 1
--Pixel Data
... . To the end of the color table
0 - first pixel on the first row
1 - second pixel on the first row
...
```

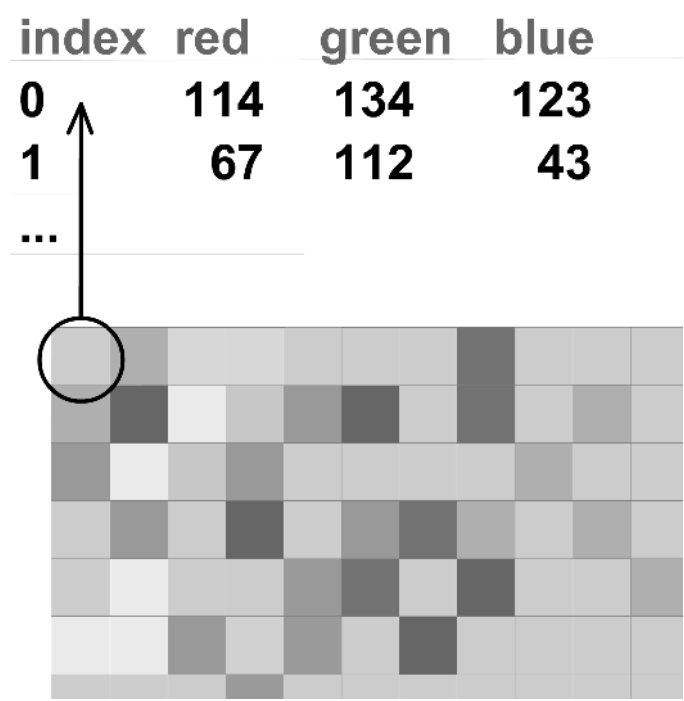


Figure 2.7 *Defining a color using a palette*

Take a look at Figure 2.7; here the pixel in the top left corner of an image can be defined using the red, green and blue values 114, 134 and 123 or we can use a color palette and then define the color by pointing at the palette index, in this case 0. Using this method we add extra information because we include a palette of up to 256 colors, or 768 bytes of information. But the pixel data only uses a single byte. Each pixel uses two bytes less than in a 24-bit image. Despite this the colors can be chosen from the same possible selection, but this is limited to a maximum of 256 different choices. By saving two bytes per pixel we only need an image with more than 384 pixels to use less storage. A 150 square image uses 22 500 pixels so we have a significant saving in storage space. Images stored in this manner are called palletized or 256-color images.

Using a fourth byte to store the opacity of the pixel can further enhance the way that each color is saved. Images stored in this way use 32 bits per pixel, eight for the red, eight for the green, eight for the blue and a further eight for the alpha.

The gif file format

The graphics interchange format (gif) is a good choice if the artwork uses a lot of flat color; the image is saved without any loss of information and when loaded it is a byte-for-byte duplicate of what was stored. The format supports a single color of transparency rather than a full alpha channel; a color is either completely opaque or completely transparent, with no half measures. The gif format can also store multiple images. It uses the same type of compression that is used by a zip or

stuffit file. The gif format is not suitable for photographic images because it is limited to a maximum of 256 different colors.

The jpeg file format

The joint photographic experts group (jpeg or jpg) format is a good choice if the artwork is photographic and file size is an important consideration. The jpeg file format is a lossy form of compression; the restored image looks similar to the original but is not a byte-for-byte copy; jpeg images cannot include a transparency setting and are either full color or grayscale.

The tif file format

The tagged image file format (tiff or tif) is a good general-purpose bitmap image format. The file can compress images using the nonlossy lzw compression method (Lempel Ziv Welch data compression and decompression technology) used in zip files or can leave the image uncompressed. It is equally suited to flat color or photographic images and can provide a reasonable compression method when file size is not very important. Because the format is nonlossy, image quality is assured. A tif file can include a full eight-bit alpha channel for professional quality overlaying.

The tga file format

Created by Truevision the tga format is often used by 3D computer animators. The file only supports the most basic of compressions, run length encoding. This compression works very well on flat color images but can sometimes make photographic images bigger rather than smaller! A tga file supports a full eight-bit alpha channel and has a guarantee of quality because of the nonlossy nature of the compression.

The png file format

The portable network graphics (png) format is the newest one on the block and was developed as an alternative to gif when the creators of the compression algorithm for gif decided to start charging royalties. It is a nonlossy format that can support high bit depths. It represents one of the most effective file formats if you want to retain full alpha-channel transparency across several applications. You will find that some application programs seem to ‘forget’ the transparency when you save an image to disk.

The psd file format

The photoshop document (psd) format includes features that make editing much easier. The most important of these is the ability to use layers. Adobe Photoshop allows the designer to have multiple layers in an image file.

The eps file format

The enhanced postscript (eps) format is used extensively in the printing industry. Although not strictly a bitmap file format it can be used to embed a bitmap image; eps files can include vector art and text information and they are rarely if ever used for sprite animations.

The ai file format

One of the most popular vector graphics tools is Adobe Illustrator. These files are saved with an ai extension. The information stored to disk is a series of instructions that the computer can use to reconstruct the image and not a series of dot color values.

Which should I use with Director?

Lots of options, so which is the best? The answer is ‘it depends’. As a general rule if your paint program supports png files, and most will, then use this format to import into Director. It is nonlossy so you can be guaranteed quality and it supports full alpha-channel transparency. You can see from Figure 2.8 that full alpha transparency gives much smoother outlines for the low resolution images you are going to use in your games.

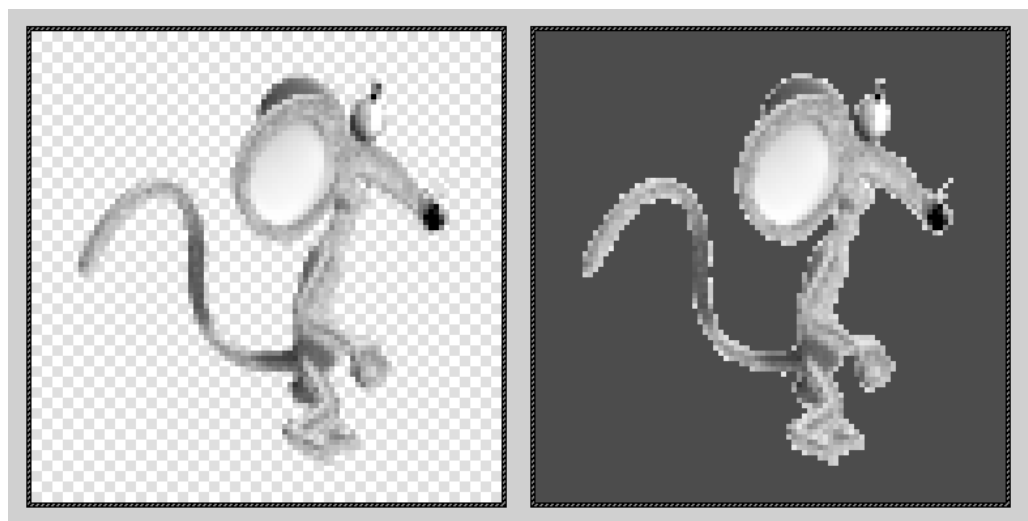


Figure 2.8 *Alpha transparency and single color*

Designing a character

Personally I don't think you can beat a real pencil for getting the feel for a character design (see Figure 2.9), but some colleagues prefer to use a tablet and draw directly with their computers. Whichever method you choose you will need to be prepared to revisit your design many times before a really good character design develops. Character design is greatly affected by the fashion of the moment. Nevertheless there are a few golden rules.

- Make sure the character is distinctive.
- Be prepared to exaggerate features.
- Keep the design simple.
- Think in color not black and white.

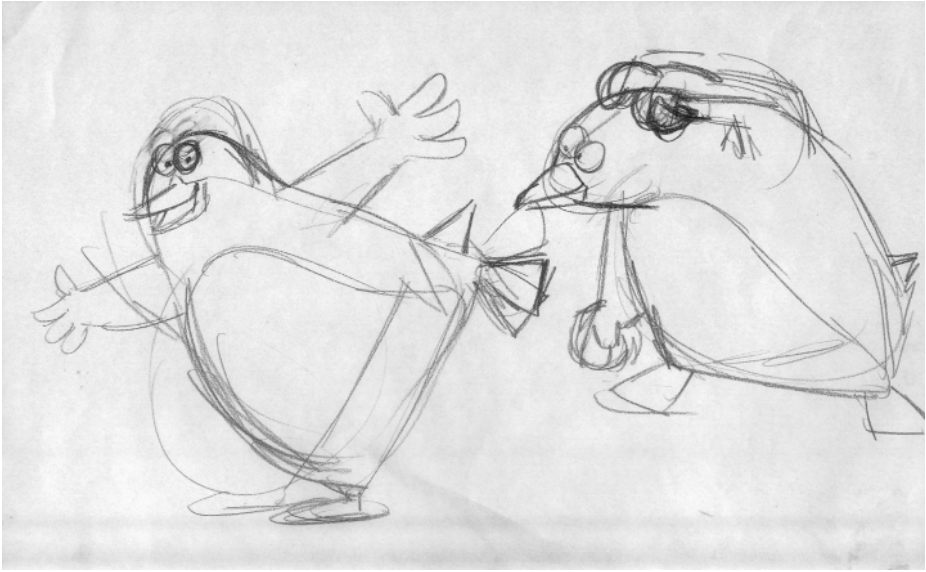


Figure 2.9 *First scribbles for a character*



Figure 2.10 *The final design*

Dramatic performance in most games is usually very limited. Your character will rarely be called on to be a great actor, but an expressive face is usually going to be an asset to a design (see Figure 2.10); it is better that your character can convey a range of emotions even if they will not be called on to prove it.

Moving between Director and your paint program

When you are working in Director you may want to alter the images that you have prepared in your favorite editor. If this is the case then you should set Director up to use an external editor for bitmap files. Choose 'Edit/Preferences/Editors...', which will launch a dialog window (see Figure 2.11). The file formats that Director understands are all present in the list shown in this window.

Highlight the file format that you want to set up and select 'Edit...'. This brings up the dialog shown in Figure 2.12.

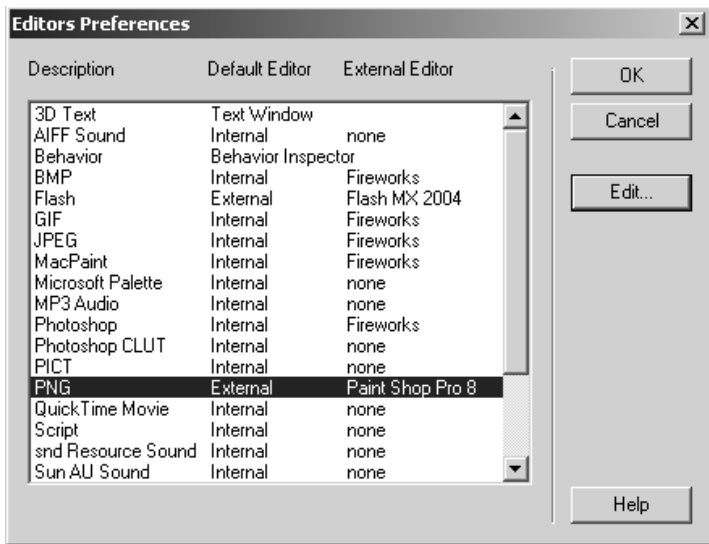


Figure 2.11 Setting up Director to use an external editor

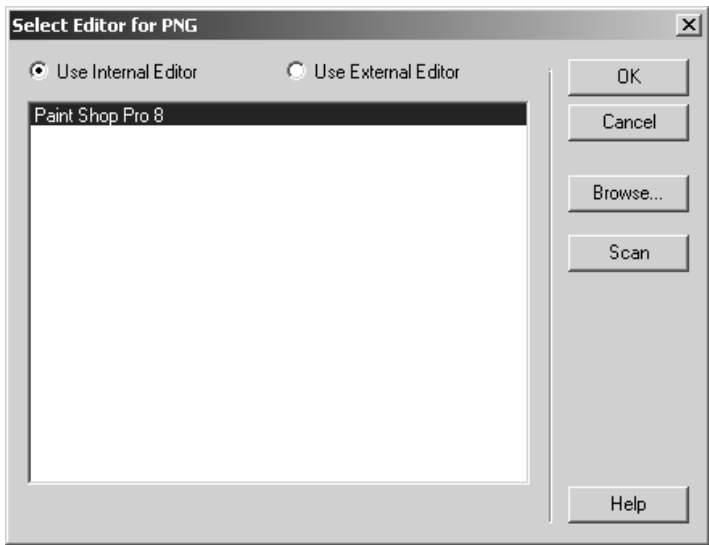


Figure 2.12 Editing details about the external editor

You can either ‘Browse...’ for the editor of your choice in which case you will need to select the actual program that you prefer to use for your bitmap editing, or you can scan for the program. With the scan option Director queries the computer for any application installed that may be chosen to edit the selected format. Then you can simply choose from a list.

Now to edit the bitmap using your chosen editor, right click on the image in the cast window and select ‘Launch External Editor’ (see Figure 2.13).

Summary

Games use lots of dynamic imagery; in this chapter you learnt some background to help you decide which way to go about producing these images for your own games. You learnt that there are many different file formats to use and the benefits and problems of some of the most common. You discovered the essential differences between bitmap and vector formats and images with palettes and full-color images. You learnt how easy it is to switch between Director and an external editing program. In most instances you are likely to use sequences of transparent png files to import animation sequences into Director. In the next chapter we will learn the basics of animation so you understand how these sequences can be created.

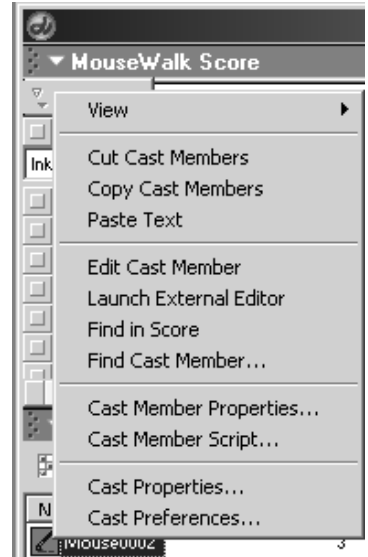


Figure 2.13 *A cast member pop-up menu*

3 The walk cycle

Animation works by showing a sequence of similar but different images in quick succession. The eye reads the result as a moving image. A great challenge when animating is to create a good walk. In this chapter in order to help to explain the principles of animation, we are going to develop a walk, from crude beginnings to a TV quality version. The animations in this chapter have all been created using Flash MX 2004; we will look at how you can export animation sequences from Flash into Director to use in your games. The principles presented in this chapter work just as well in alternative applications but Flash is a good tool to get familiar with; if you do not currently have a copy then you are recommended to download the trial version from www.macromedia.com/downloads, which gives you the full version of the product for 30 days.

Segmenting a character

Cutout animation is produced by splitting a character into small segments and then animating the segments simply by moving and rotating (see Figure 3.1). If you intend to use motion tweening, a very fast method for moving and rotating symbols in Flash, then you will need to place each segment on a separate layer. In this chapter we will take a very simple cat character and create several walks of increasing sophistication. The character was drawn directly in Flash using the drawing tools available. Then the character was divided into sections using the lasso tool. To select the lasso tool you need to click the 'Tools' button indicated in Figure 3.2.

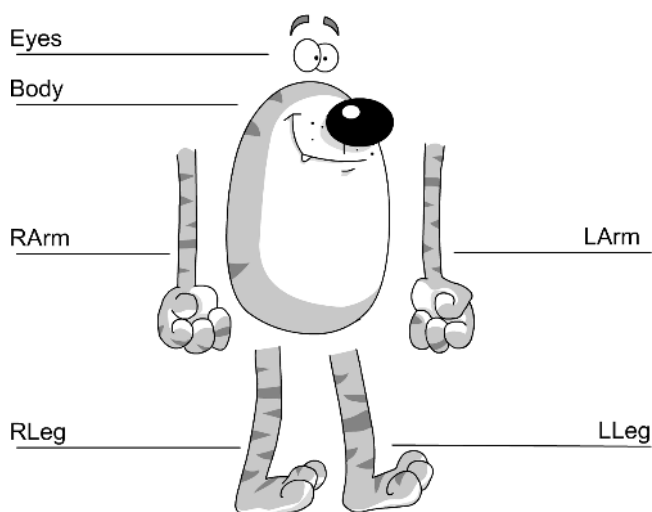


Figure 3.1 *The segments in the simple walk*



Figure 3.2 *Selecting the lasso tool in Flash MX 2004*

To use the lasso tool draw around the character by clicking and holding the left mouse button. The selected area will show an overlaid dot pattern (see Figure 3.3). To convert a selected area into a symbol press 'F8' or choose 'Insert/Convert to Symbol...' from the menu. In the dialog box choose 'Graphic' as the symbol type and enter a descriptive name. The selection will change from the dot pattern to a bounding box. The name that you give the

symbol will be the name displayed in the 'Library'. You can view the Library at any time by pressing 'F11' or selecting 'Window/Library' from the main menu. When the segment is a symbol you can use it as many times as you like either by dragging it out of the Library or by copying an instance of it on the stage and then pasting the instance into a new frame or new layer of the current frame. You can even paste a duplicate of the symbol in the current layer but remember that this will not work with motion tweening. The next step is to ensure that the rotation center is correct.

If you intend to use motion tweening then it is important that the point about which the segment will rotate is carefully chosen (see Figure 3.4).

To adjust a symbol's pivot point, select the 'Free Transform' tool from the 'Tools' palette and move the white circle. The white circle defines the point about which the symbol will rotate. To get an accurate placement it is usually best to use the 'Zoom' tool. Using the 'Zoom' tool you can move in and out by selecting either the '+' option or the '-' option and clicking, or you can drag a rectangle and then Flash will resize to suit this rectangle. The amount the artwork moves when using the arrow keys is dependent on the scale of your view of the artwork, because the pixel movement refers to screen size not the original size of the artwork. Move the white circle until the rotation center is in the most appropriate place. It is important that you understand the difference

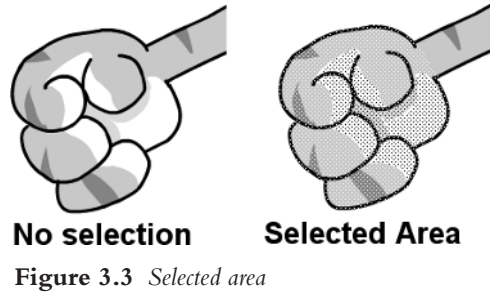


Figure 3.3 *Selected area*

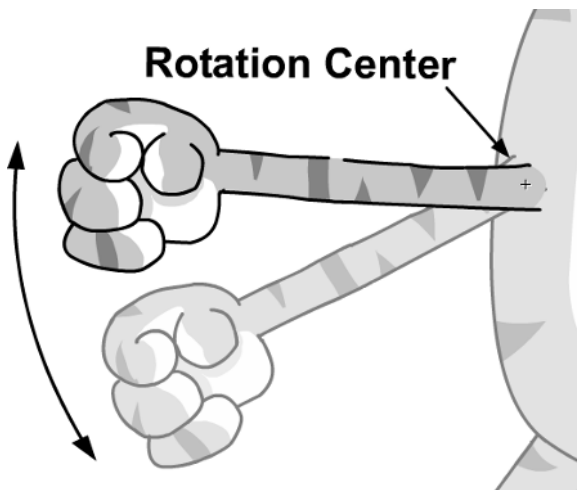


Figure 3.4 *Selecting the rotation center*

between the positioning of the artwork inside a symbol in relation to its rotation center and the positioning of a symbol on the 'Stage'. This difference is very important. You can move the rotation center so that it is a long way from the artwork inside a symbol, but the problem is that when you start to rotate the symbol it will stray away from where you intend it to be, consequently you will need to add multiple keys to get it back in place. These multiple keys negate the advantage of using motion tweening. If you find you have to use a great number of keyframes because of problems occurring when you rotate a symbol then it is often best to check the location of the pivot center using the Free Transform tool.

A first walk with this character

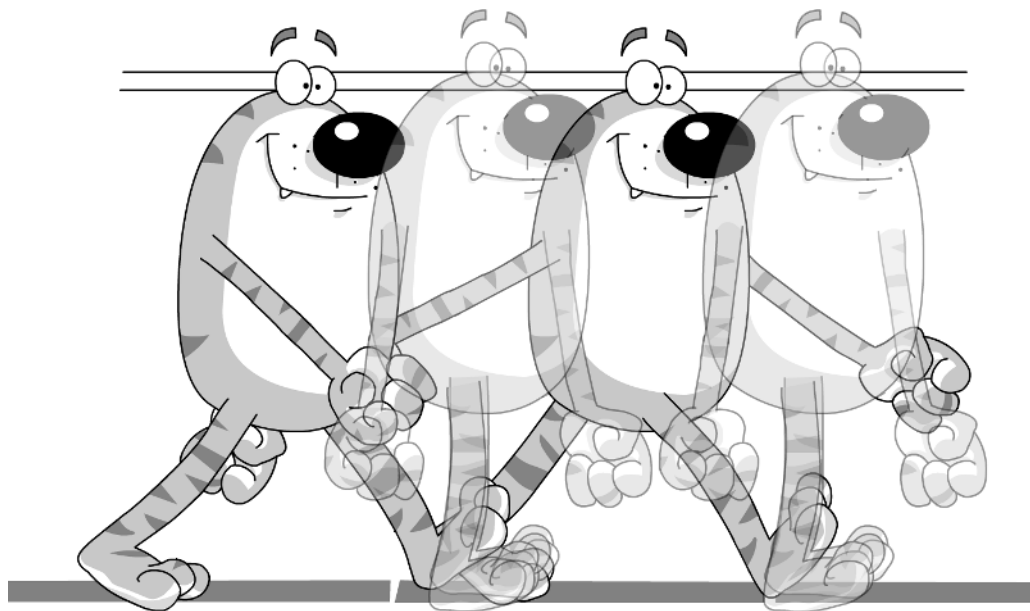


Figure 3.5 *The principal keys in the simple walk*

See Figure 3.5. Now is a good time to open the project 'Examples/Chapter03/FatCat.fla'. Take a look at 'Scene 1'. Recall that you can select the current scene using the 'Edit Scene' button at the top right of the timeline or by using the 'Windows/Panels/Scene' panel (see Figure 3.6).

Take a look at Figure 3.1 and you can see that the segmenting of this character is very simple. Each segment appears on a new layer inside a new symbol called 'Walk1'. On the root timeline there are keys for the symbol Walk1 at 1, 7, 9, 15 and 17. These ensure that the cat goes up and down as it walks. Because all the limbs are inside the symbol Walk1 these will go up and down with the body so the arms and legs will not become separated from the body as it goes up and down. Nesting symbols inside other symbols in this way can make your animations easier to prepare. But before we put in the keys for the main Walk1 symbol we need to prepare the main key positions.



Figure 3.6 *Selecting the current scene*

Figure 3.5 shows the extreme positions for the cat. The left leg is fully back at frames 1 and 17, while the right leg is back at frame 9. The arms are opposite to the legs; when the right leg is fully forward the right arm is fully back. Always refer to your segments in relation to the character. The right arm should always be the character's right arm. If the character is facing you then it is tempting to call their right arm the left arm, because it is on the left in the screen view. But at some stage the character will be viewed side on or from the rear, so at what point would you swap? At least if the segments always refer to the character's right side then you can work it out and it becomes less confusing. Make the key positions that you have entered into 'Motion' either by selecting the frame in the timeline and choosing 'Create Motion Tween' from the context menu or by selecting Motion from the 'Tween' combo box in the context-sensitive 'Properties' panel for a frame. The character is now animated. Make sure that 'Control/Loop Playback' is checked in the menu and play the animation using the 'Play' button, 'Control/Play' from the menu or by pressing the 'Enter' key. Because you are inside the symbol the cat's legs and arms swing but the body does not go up and down. Go back to the root timeline either by double clicking in an empty space or by clicking on 'Scene 1' at the upper left corner of the timeline. To complete this animation, adjust the height of the 'Walk1' symbol at frames 1, 7, 9, 15 and 17 so that the feet stay on the floor. Make frames 1, 7, 9 and 15 Motion-type frames and play the animation. It is very stiff but at least it has started to move in a very basic way. You can create a walk this way very quickly and if you do a 'Control/Test Scene' you will see that the file size for this scene is only 6 K, not bad for an animation.

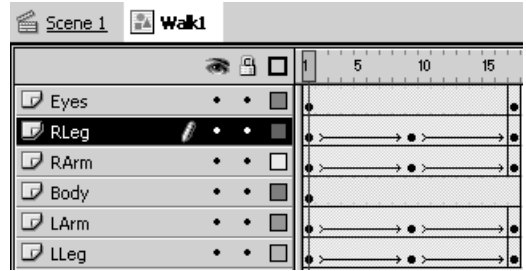


Figure 3.7 Using motion tweening for this animation

Improving the walk

To improve the walk the arms and legs will need further segmenting. At this stage the benefits of nesting clips inside clips start to be outweighed by the difficulties of actually editing the animation. The segments we are going to use are shown in Figure 3.8. Each of the named segments is a symbol in the 'Library'. The layers used have been reduced somewhat because we are not going to use any motion tweening, we are going to add all the keys individually (see Figure 3.9). Take a look now at 'Scene 2'.

Because all the positions are going to be individually animated this animation will take a little longer to do. First, set up the extremes; the stretched leg positions at frames 1, 9 and 17 are simply 1 repeated because we want the animation to loop. Second, put in the passing positions at 5 and 13. The body will be higher at the passing position (see Figure 3.10).

Then add in the down positions at 3 and 11 and the up positions at 7 and 15.

Finally, do all the in-between positions using the onion-skinning technique. To use onion skinning, click on the onion-skinning button in the lower left of the timeline (see Figure 3.12). Select the range of the onion skinning by moving the circle handles in the onion-skinning range

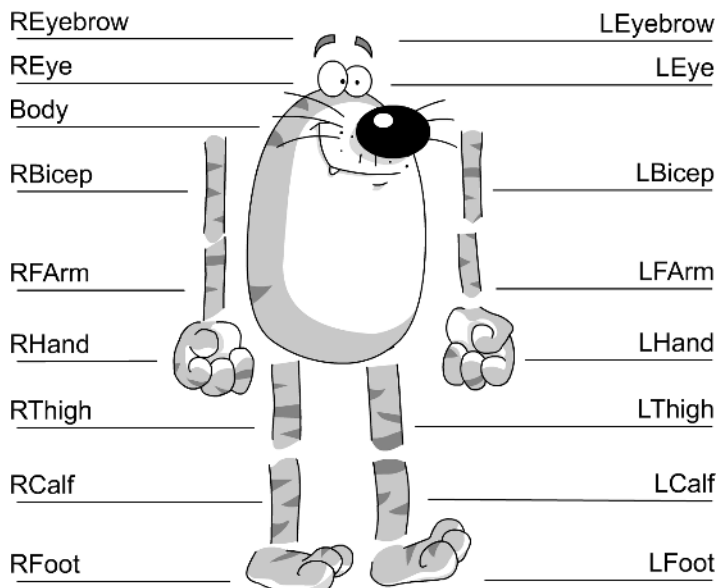


Figure 3.8 *Extra segments in the developed walk*

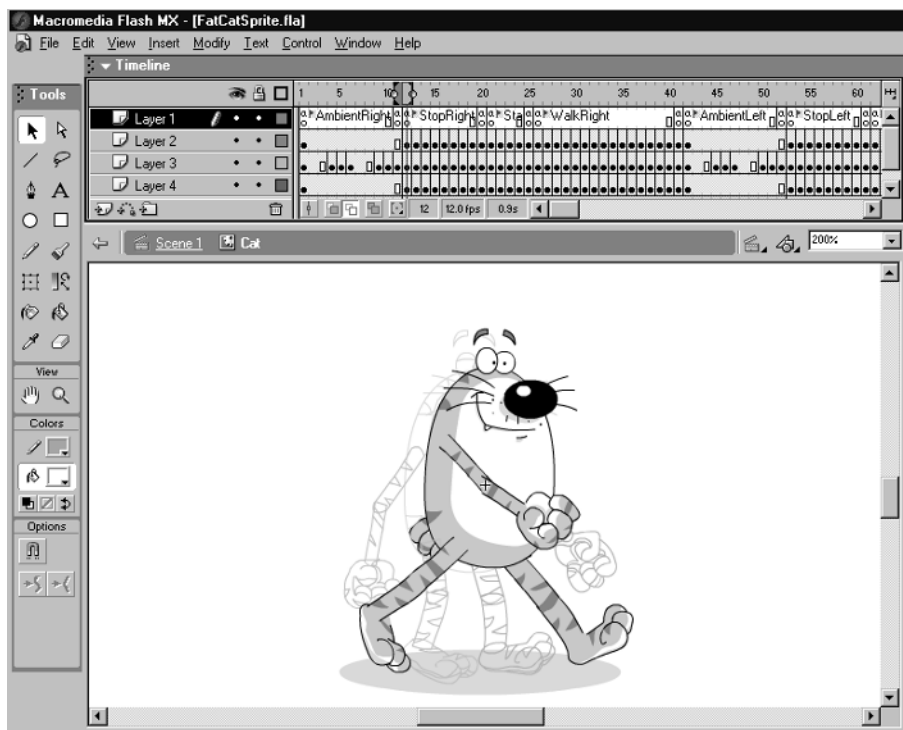


Figure 3.9 *Layers used for the animation*

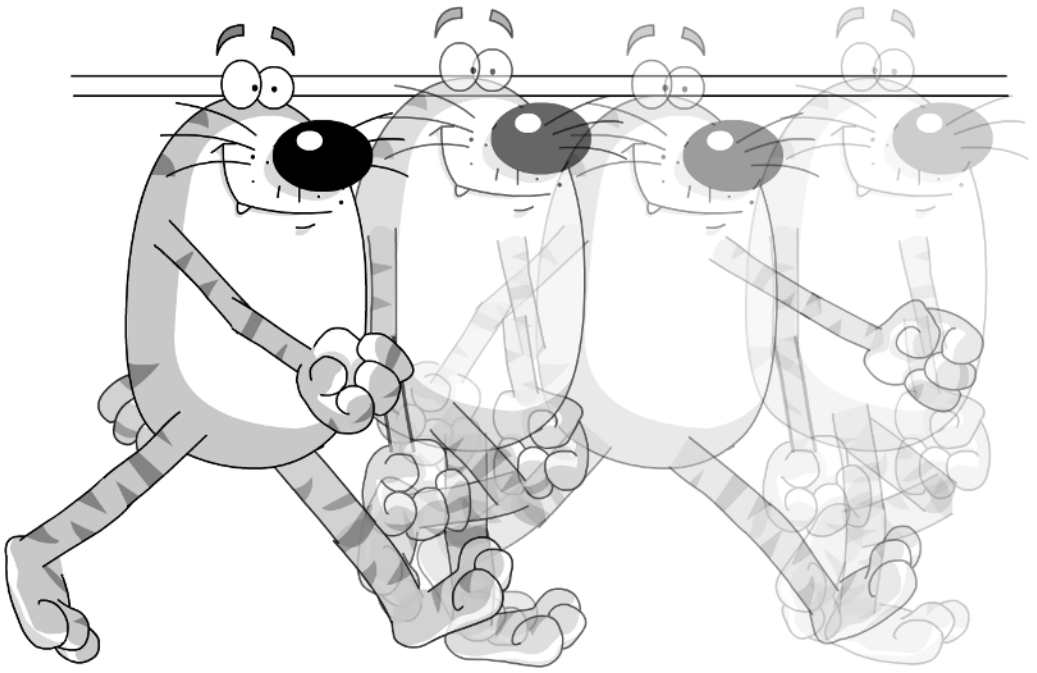


Figure 3.10 *The extremes and the passing positions*

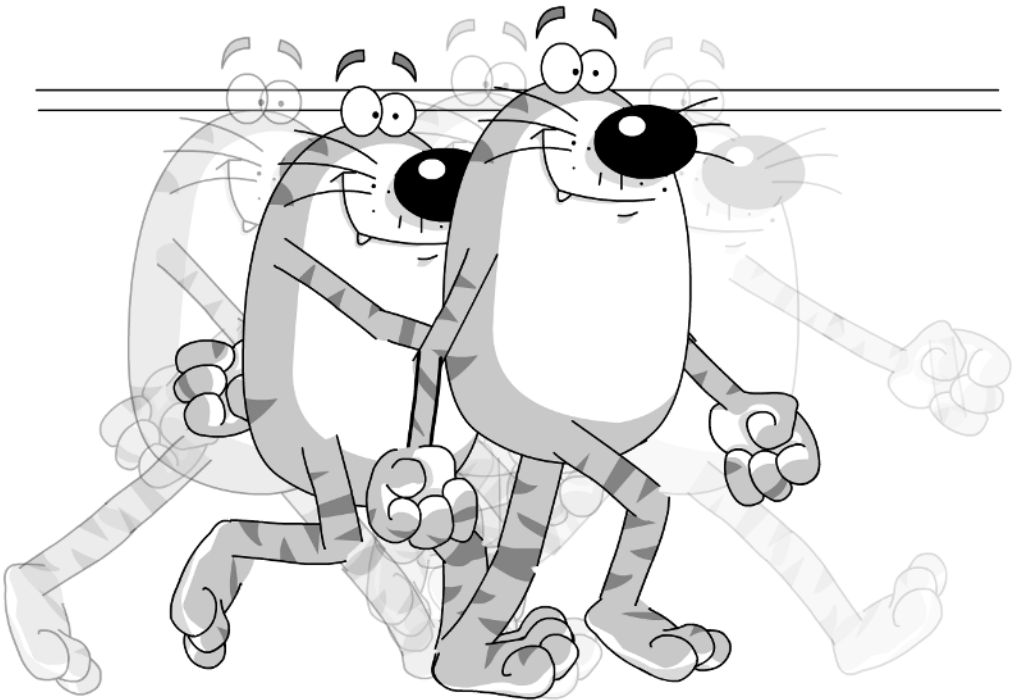


Figure 3.11 *Adding the up and down positions*

indicators. Onion skinning allows you to see the frames that precede the current frame and those that follow it. The fewer preceding and following frames that you view, the easier it is to work out what is happening. For these final in-betweens you need only view one preceding and one following frame. To do the in-betweens start by just moving the body, feet and hands. If you have difficulty moving a segment because it is behind another, lock the offending layer using the 'Lock' button in the timeline options.

Before you move the thighs, calfs, biceps or forearms switch off onion skinning. You will have a much better view of the artwork by then and the position of the body, feet and arms will dictate where the other segments need to go. 'Scene 2' through to 'Scene 5' show various stages of creating this walk. I hope you agree that it is much looser than the first walk. This is a standard walk but a standard is made to be broken. 'Scene 6' shows how the walk can look by making the passing position a low position. The effect is to create a walk with a double bounce; it is a popular technique for many cartoon characters. Try to adjust the walk to get as much life and vitality as you can.

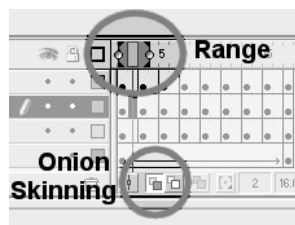


Figure 3.12 *Selecting onion skinning*

Exporting the animation to Director

Now we have an animation we are happy with we want to get this into Director. First we need to know roughly how big the character is going to be on screen in the final game. For this example we will have a sprite which is 150 pixels square. We find the easiest way is to create a new Flash movie the same size that you want the sprite. If the sprite has a dark outline then set the background color for the movie to a dark gray. Simply click on the movie area and use the color box in the 'Properties' window to set the color.

The purpose of this is to ensure that the anti-aliasing does not leave awkward looking pixels around the outside. Using this method we will have a full alpha channel image. To make sure the image looks smooth Flash uses anti-aliasing. If we look very closely at the edge then you can see that it is made up of lots of different colors.

These colors are an attempt by Flash to blend into the background color. The best all-round choice is a compromise between the outline silhouette of the character and the main colors of the background that the sprite is going to be viewed against.

Create a new symbol in Flash and call it 'Cat'. Copy and paste the animation frames from the final version of the walk into this new symbol. Open the 'Library' panel by selecting 'Window/Library' and drag the Cat symbol onto the 'Stage'. Go to the timeline and click the mouse on frame 16; press 'F5' to get a strip between 1 and 16. Finally size the symbol so that the animation does not go



Figure 3.13 *Changing the background color in Flash MX 2004*



Figure 3.14 *Sprite detail on white background*



Figure 3.15 *Sprite detail on dark background*

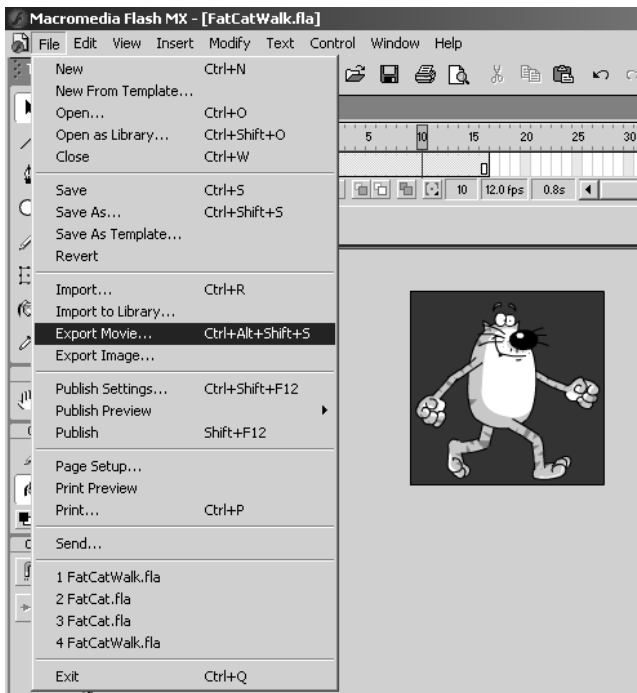


Figure 3.16 *Exporting the frames as a png sequence*

off the screen. If you press 'Play' on the control bar or press 'Enter' then you should see the Cat take a step. To see the animation cycle, select 'Control/Loop Playback'. If you are happy with the results then select 'File/Export Movie...' (see Figure 3.16).

This opens a new dialog with several options. Select 'PNG Sequence' and choose a base name and location for the image files (see Figure 3.17).

Having pressed ‘Save’, you are now offered some additional options (see Figure 3.18).

For ‘Dimensions’ and ‘Resolution’ simply hit the ‘Match Screen’ button. Choose ‘Full Document Size’ for the ‘Include’ option, which just ensures that the images are the size and shape of your movie and do not include bits that go out of screen. For ‘Colors’ choose ‘24 bit with alpha channel’ and you do not want a ‘Filter’. Finally choose ‘Smooth’. When you have done all this hit ‘OK’ and Flash will create a series of bitmap images.

Now open Director and select ‘Import...’. Highlight all the images and press ‘Import’ in the dialog box (see Figure 3.19).

Another options box will appear (see Figure 3.20).

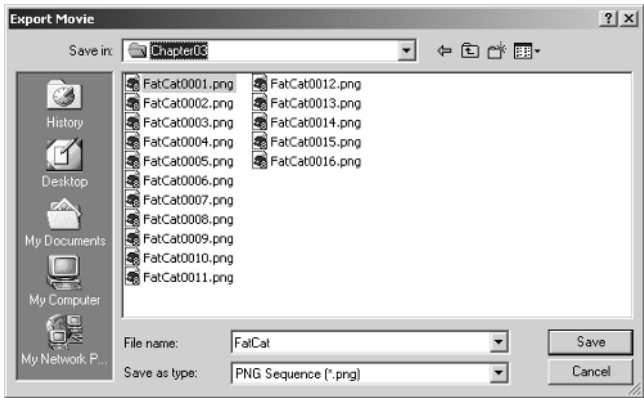


Figure 3.17 Choosing where to save

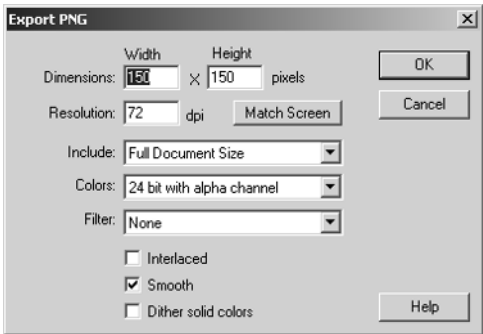


Figure 3.18 The png export options

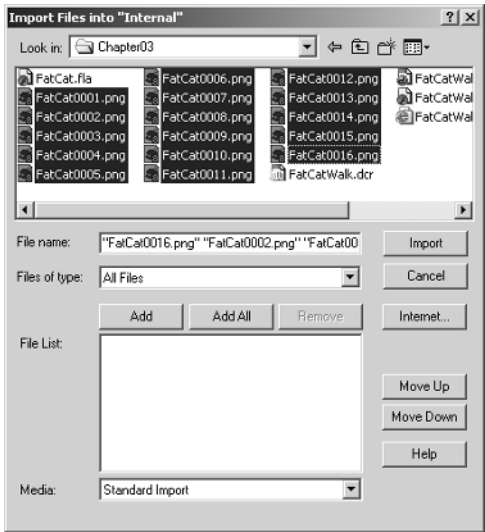


Figure 3.19 Importing files into Director



Figure 3.20 *Selecting the import options*

To use the alpha channel successfully select 'Image (32 bits)', unselect 'Trim White Space' and check 'Same Settings for Remaining Images' then press 'OK'. The images are loaded into the 'Cast'. At this stage you can only see them if the Cast window is visible. Open the Cast window if it is not already open and highlight all the images. Now choose 'Modify/Cast to Time'. The images are added to a single channel in the score, from wherever the cursor was last set (see Figure 3.21).

Because we have a full alpha channel we can just use the default setting of 'Copy' to get a full transparent sprite on the stage (see Figure 3.22). Press the play button to see the animation loop.

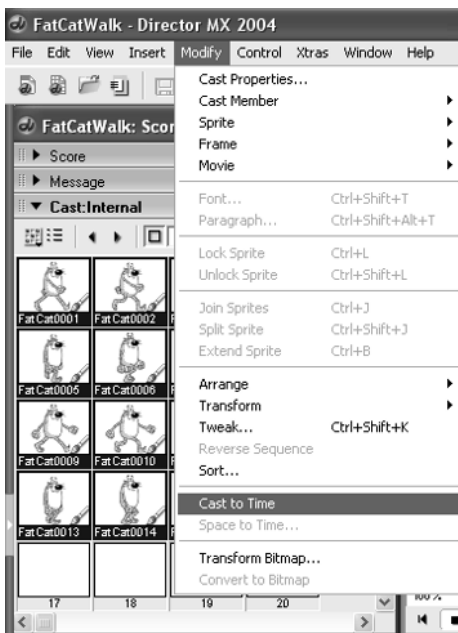


Figure 3.21 *Placing the images on the 'Stage' using 'Cast to Time'*

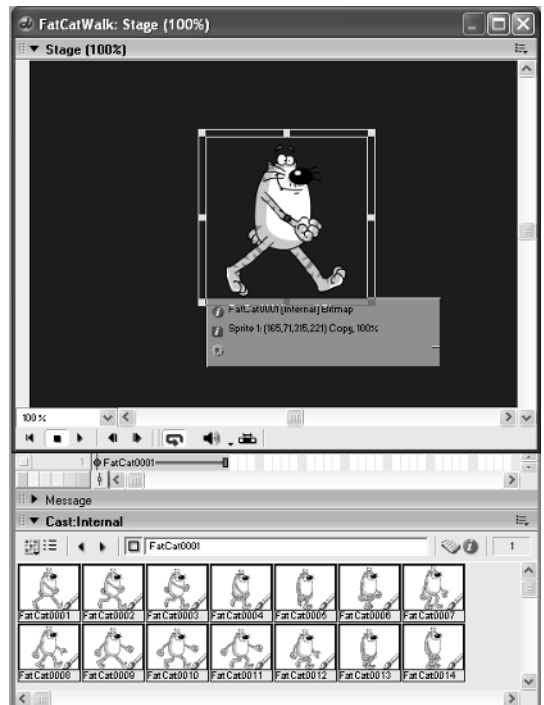


Figure 3.22 *The imported files in the 'Cast' and on the 'Stage'*

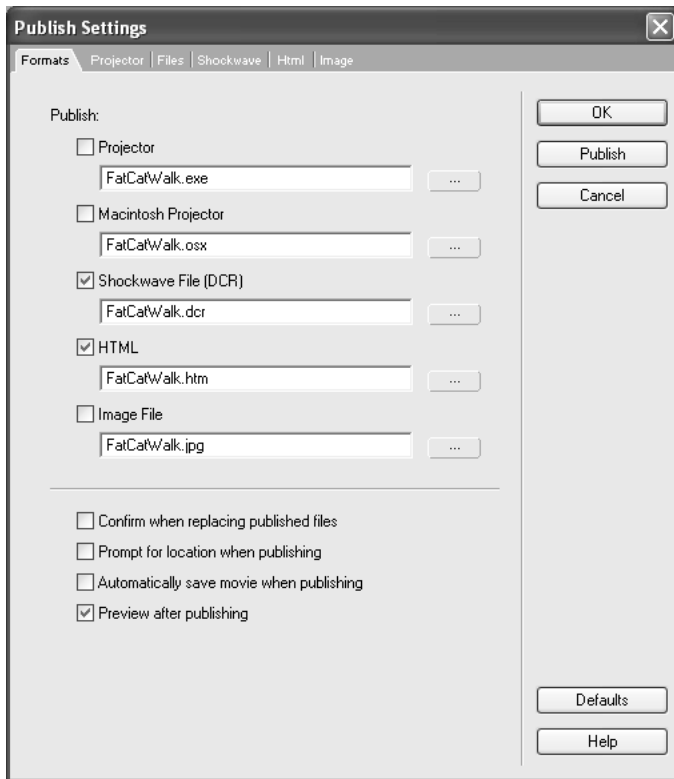


Figure 3.23 *The 'Publish Settings' dialog*

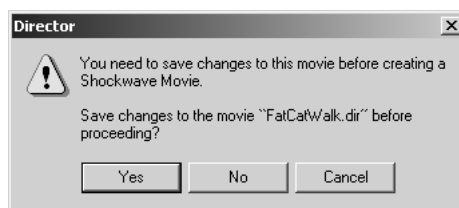


Figure 3.24 *Saving the movie before publishing*

It just remains to publish this in a web-friendly way. Select 'File/Publish Settings...' to open the dialog box seen in Figure 3.23.

For the purposes of this example just accept the default settings. To publish choose 'File/Publish'.

You can now view the file in your web browser (see Figure 3.25).

The final animation is a web-friendly 96 K in size. The movie files and the final exported result are all available on the CD in 'Examples/Chapter03'.

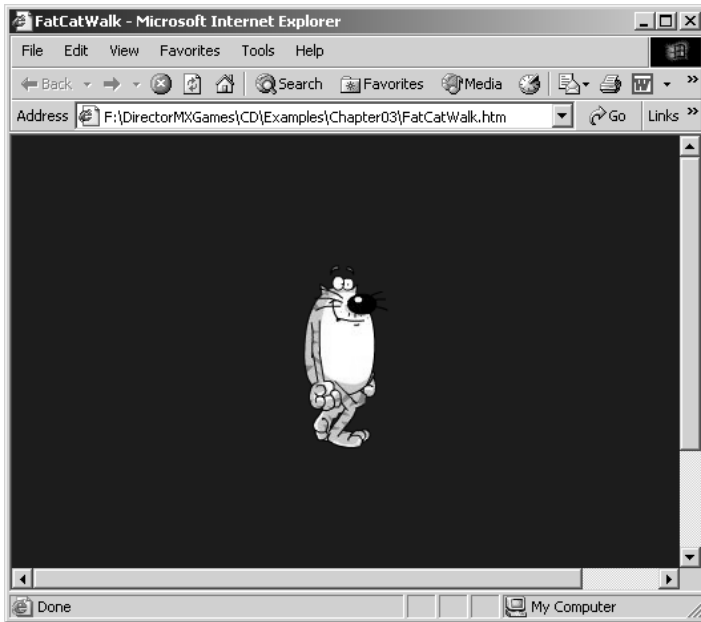


Figure 3.25 *Seeing the result in a web browser*

Summary

In this chapter we looked at how to create interesting and dynamic animation using the popular tool Flash MX 2004. We looked at creating this using motion tweening and using keyframes. We looked at how to enhance a cutout animation using additional drawn frames. We looked at how to export the resulting animation to Director and how to set this up on the Director score. Finally we looked at publishing the resulting animation in a web-friendly form.

4 Background art

All your games will require some kind of background artwork. Your animated characters will need to appear in some context. In this chapter we look at creating backgrounds to act as a setting for the action that takes place in your games. There are many options for creating the background artwork, most of which are aesthetic, but some are pragmatic. For example, there is no point creating a fast-scrolling game that will not scroll fast because of the complexity of the background artwork. Even if you have no intention of creating background art you will find the tips in this chapter useful.

The benefits of keeping the design simple

Design is such a subjective thing. One person may like something that another finds horrible. Design is connected with fashion; the art and design of 30 years ago differs markedly from the art and design today. But there is one thing that persistently holds true and that is – simplicity is attractive. Complex imagery can be fascinating but the understated usually wins out. Warner Brothers' cartoons often had the barest of backgrounds but they were just enough to give context without detracting from the main focus of a scene. All the images in this chapter can be seen in the project file 'Examples\Chapter04\bgs.dir'. Many of the images look so much better in color, so you are recommended to look at the artwork in order that you are better able to judge the comments about each example. The first image we will consider is the simple tree shown in Figure 4.1. This

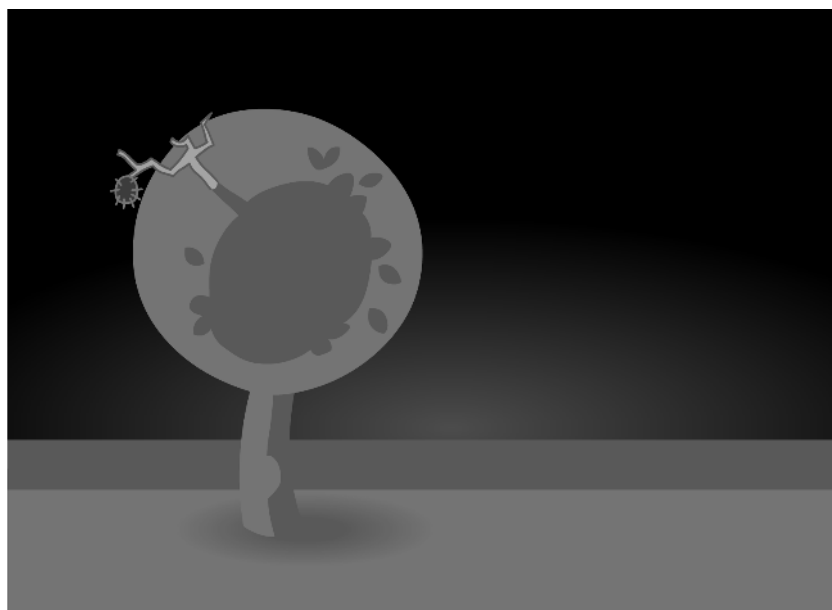


Figure 4.1 *A very simple tree*



Figure 4.2 *Moon and stars*

design by Christian Holland is a perfect example of the attractive quality of simplicity. Just simple flat color and the barest suggestion of leaf structure say ‘tree’ instantly but with style and charm.

Figure 4.2 by Suzie Webb shows the surface of the moon in four colors and again just the hint of detail. The design has just enough form to evoke the place yet is delightfully understated. Simplicity is so often the most effective option.

Figure 4.3 shows another design by Suzie, which again, in just a few bold colors, creates a terrifically sunny bedroom, with a modern yet retro 1950s feel. That is another aspect of design; a clever designer can take influences from another time and place and create artwork that has a feel of the original while being totally new and modern at the same time.

Creating games is a complex technical challenge but the user sees the design first before they are captivated by the game play. Most designers assimilate the influences around them. It can be painting and the traditional arts but it can just as well be a cooker or a chair that catches their attention. Comic book art, children’s books, advertising, films, toys, newspapers, photography; the inspiration is out there. If you are interested in design then arm yourself with a camera and capture images that inspire you. This helps you remember and may form the catalyst when faced with a new design project.

Using a keyline

Traditional comic book art uses a keyline. The reason for this was convenience rather than design; the artwork was often created with a team of people and an artist, colorist and inker. The



Figure 4.3 *A simple bedroom*



Figure 4.4 *Incorporating line in the design*

colorist could keep to a handful of simple colors and the inker pulled out the detail using a keyline. Figure 4.4 shows an image design by John Ashton. The bendy nature of the design is a classic cartoon signature.

Figure 4.5 takes the use of a keyline to perhaps the ultimate conclusion.

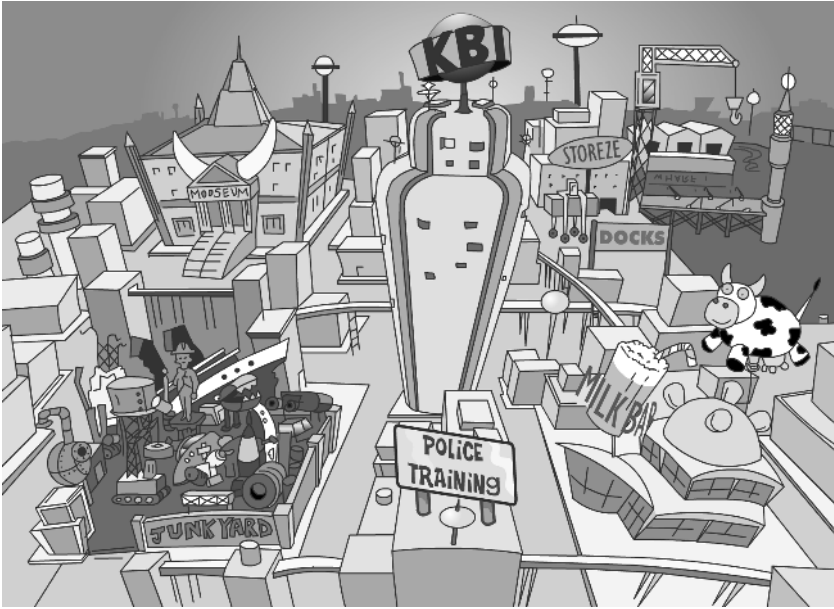


Figure 4.5 *A complex background*



Figure 4.6 *Give your characters space*

When you create your background art make sure that it does not overshadow your characters, which will need to have the space to move around and should not be overconfined by the design. Also consider using several layers so that your characters can appear from behind a section of the background. Figure 4.6 shows a simple medieval room set; the design allows the central magician

character the room to walk up the stairs disappearing behind the wall as he climbs the spiral steps. The design incorporates the chimney for the magician's cauldron. The character goes behind this element of the set giving extra depth to the animation.

Importing scanned artwork and artwork created in other paint programs

Remember that you are not limited to creating artwork using a computer. Figure 4.7 shows a background that has been hand painted with watercolors. Not a computer in site! The final result was scanned and imported as a bitmap. This technique has the feel of a children's book illustration and gives a texture to the graphic that is sometimes missing in computer art. Of course there are numerous paint programs that allow you to create textured graphics directly on your PC if you prefer.

Using abstract imagery

Another interesting technique for creating backgrounds is to use abstract imagery or imagery derived from type. Figure 4.8 shows the background of a game that featured the pop group Steps. At first glance you may well have missed that the shapes at the top of screen form the word 'steps'. A second look makes it easy to see the letters. The graphic that forms the word was used repeatedly with a fading effect to create the remainder of the background design.

When creating such artwork in Flash the advantage of using a single symbol is that the file size will be reduced and the overall design is strengthened by the lack of complexity. Using type symbolically you can very quickly create lots of background designs.



Figure 4.7 *Using a scanned painted background*



Figure 4.8 *Using an abstract background design*

Building complex art using simple gradients

Most art packages have the ability to create gradients and these can be very useful when creating a background. Figure 4.9 shows the inside of a student's fridge. Students are notorious for avoiding housework. In this illustration, it is years since the fridge was given a healthy spring clean. As a result of years of neglect the fridge is beginning to have a life of its own. Mould and curry splatters are mixed with spilled bean cans and other monstrosities. Creating this disgusting scene required lots of research! Combining the same simple gradients several times, stretching and altering their opacity, created the mould growing at the edges. It doesn't seem possible that this effect was created using just radial or linear gradients.

Figure 4.10 shows another clever use of gradients; this evocative background uses a single cloud graphic and a single pole graphic, which are flipped, squashed and stretched to form the mysterious landscape. The background was used for a fun e-mail game that features two Ninja warriors facing each other in a deadly game. It is one of many great games to be found at mohsye.com.

Using computer graphic packages to create the background art

Another very effective way to create background art for your games is to use a computer graphic package. The image in Figure 4.11 shows a simple set created in the Lightwave 3D package. The image is prerendered to the correct size for the game then imported as a bitmap file. 3D packages are great for putting together background designs or buttons. You can very quickly create detailed environments using just a few interesting textures and some simple meshes. If you are regularly creating artwork for games then you are advised to spend some time learning a 3D package. After

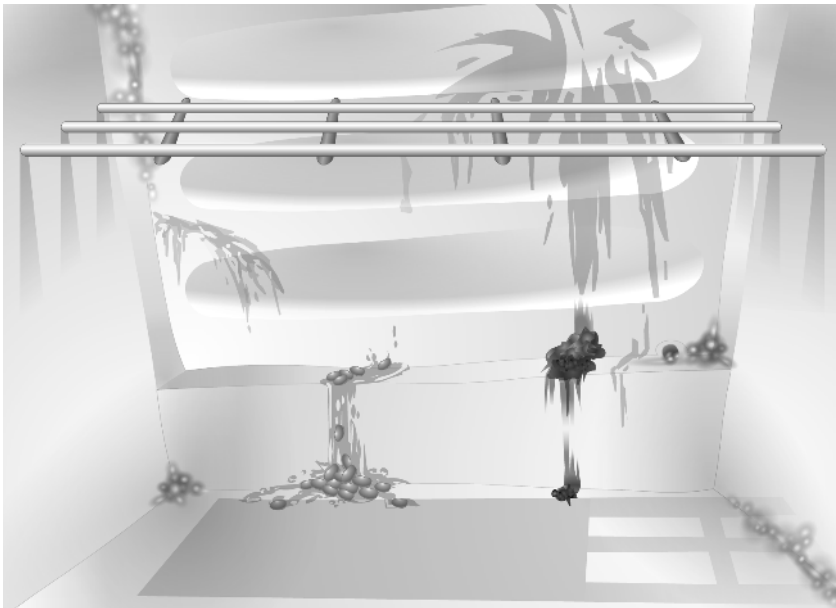


Figure 4.9 *Using gradients to add texture*

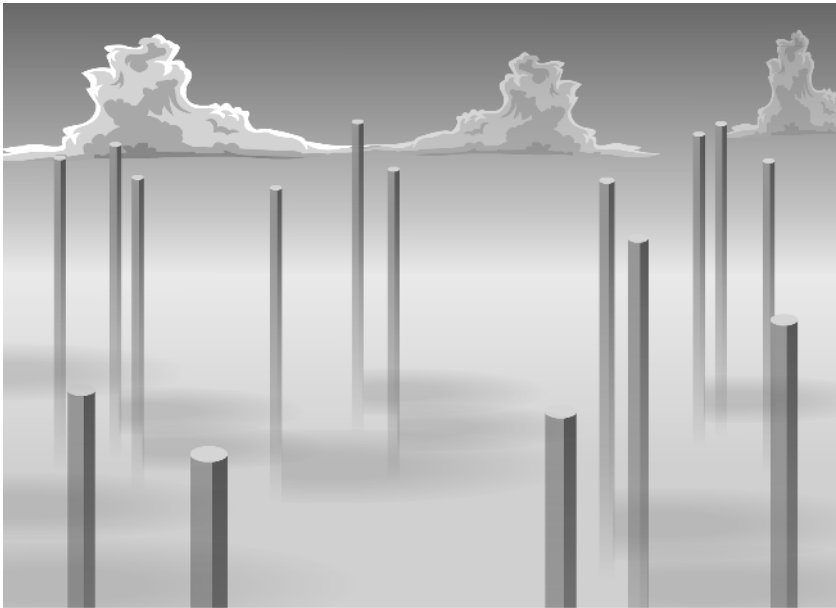


Figure 4.10 *Duplicating elements for convenience*



Figure 4.11 *Using a computer graphic background*

meddling with a few different packages we settled on Lightwave, which provides the best combination of modeling tools, animation and rendering at a reasonable price.

Using tiled backgrounds

If your background uses repeating elements then consider creating the background as sections and importing each section as a png bitmap. The advantage of the png format is that it is lossless and can incorporate an alpha channel. The box shows a summary of the file format options we looked at in Chapter 2.

Bitmaps come in many different file formats. A Director developer is likely to use jpeg, bmp, gif and png files. The eps format is not a bitmap, instead it is a vector format, but an eps image can contain a bitmap; this is not recommended because the bitmap image is resolution dependent while the eps is regarded as resolution independent. Remember that an eps is only truly resolution independent when it only contains vector elements. Here are the advantages and disadvantages of different file formats used for importing and exporting image data.

Format	Advantages	Disadvantages	Flat color	Half-tone
jpg	Small files	The compression method loses detail from the image. Can have strange speckled artefacts	Likely to create strange artefacts	Best for photographic images
bmp	Big files	Only the most basic of compression methods	Reasonable choice; bmp images can have 16 256 or 16 million colors	Will result in a very big file

png	Uses a clever compression that loses no information in the image. Can incorporate an alpha channel	Compression method will not create files as small as jpeg file	Not the best choice	Good choice for images that need to be maintained at optimum quality
gif	A relatively small file size can be achieved without losing information in the image	Cannot use 24-bit palette, the maximum number of different colors in a gif file is limited to 256	Best choice for flat color images that need a high level of compression	Not suitable
eps	Vector format ensures resolution independence. An eps can contain a bitmap; the bitmap will not be resolution independent	Not suitable for bitmaps	The best choice if the image is in vector format	Not suitable



Figure 4.12 *Using photocompositing to build the background*

The image in Figure 4.12 was created by first creating some brick tiles, the horns, the round window and the arch. Each element was imported as a png and combined to create the overall background. This results in a smaller file size and allows you as a developer to create different backgrounds using a limited number of tiles. Tiled backgrounds were very popular when computers had more limited facilities and are not used as often these days; they do offer flexibility and small file sizes so they should be considered for some game formats.



Figure 4.13 *Using a photograph*

Using photography

Sometimes the simplest way to create a great background is to go outside and photograph the real world (see Figure 4.13). For some games this can give a quick-to-produce and effective result.

Summary

There are many ways to add a background to your game. You can create the background entirely in your favorite paint package, draw or paint it or use a 3D modeling and rendering package. In this chapter we looked at some of the options available and examined why certain approaches may not be effective for particular games.

5 Using computer-generated imagery programs to create animation

In Chapter 2 we looked at using Flash to create animations. Although Flash is a great tool for creating and editing drawn animation, another method for creating animations for Director games is to use computer-generated imagery (CGI). Teaching the techniques required to use all the many computer animation programs is way beyond the scope of this book, but the principles common to all programs are briefly covered in this chapter. Along the way we will model a 3D character and create a couple of test animations to show how quick, flexible and effective this technique can be.

Character animation using a computer animation program

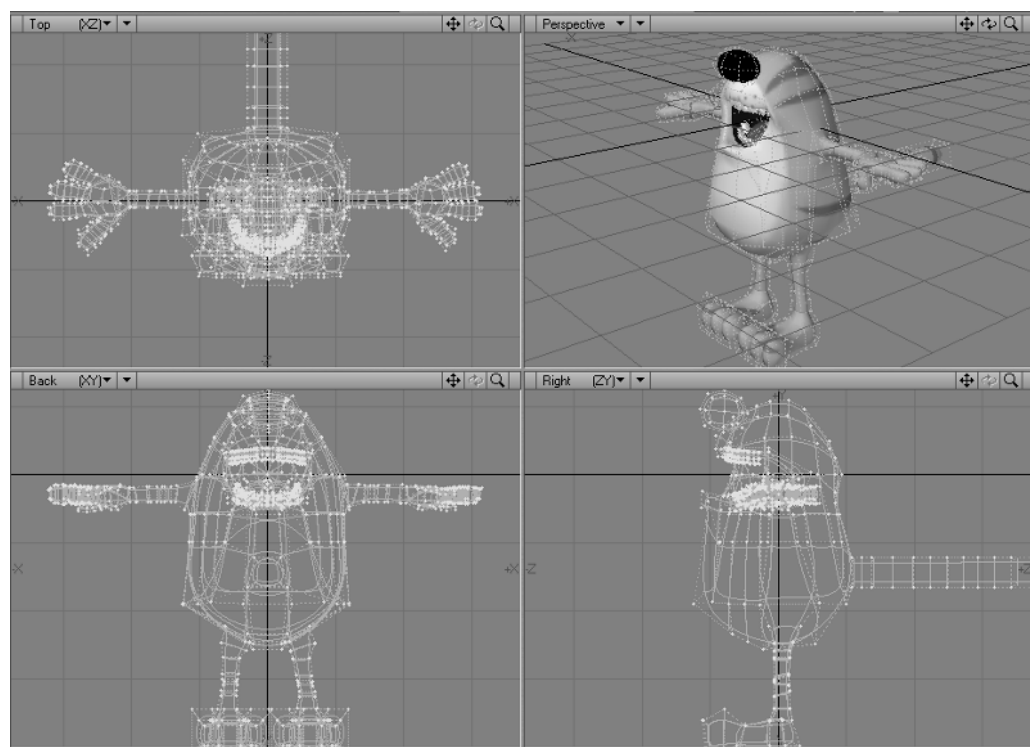


Figure 5.1 *Fatcat in Lightwave 7*

Since 1985 I have been the managing director of an animation company in Manchester, UK, called Catalyst Pictures. At Catalyst we have been using Lightwave for character animation since 1998. The reason is totally pragmatic; it is the cheapest program to offer all the facilities needed for TV quality character animation. Over the years I've even developed a few plugins for Lightwave. As you no doubt gather I am something of a CGI fan. Having spent the best part of 12 years producing drawn animation for clients who seem to like nothing better than changing things for no reason, CGI is an absolute blessing. If the character changes color or has to move faster or should be viewed from the left, right, top or bottom then the CGI work has the blessing of being constantly editable. This is in marked contrast with drawn animation where almost all changes mean starting from scratch. The character that you see in this chapter was created from a clean sheet and animated in a morning by one person. There is no way that a drawn character could ever be done in that time. Later in the book we will look at creating real-time 3D games using the facilities in Director MX 2004; the examples in this chapter are all suitable for use in the real-time environment so the techniques are very important ones to understand fully. OK, let's look at the process.

Modeling with a computer-generated imagery program

Computer animation is a three-stage process. First you create a virtual model of the character that you are going to animate, then you animate and finally you render. When modeling, most CGI programs show four views of the model, as shown in Figure 5.1. These days interfaces are all so configurable that you can usually set up the display to show you whatever you like but the default is to show a top view of the character in the display to the top left, a front view in the bottom left, a 3D view in the top right and a side view in the bottom right. Different programs approach modeling in different ways. A few years ago there was a craze for Non-Uniform Rational B-splines (NURBS) modelers. This craze has waned because NURBS modeling is extremely hard to do for arbitrary meshes. What is an arbitrary mesh? It is a mesh that has points with sometimes six lines meeting, but sometimes more or less. A totally standard mesh of triangles would have six lines meeting at every point (see Figure 5.2).

The new craze is for polygon modelers that use subdivision surfaces. A polygon model is the simplest type of model that you can have in CGI. Essentially to create a polygon model you join points in 3D space to create polygons; usually you will create either three- or four-sided polygons. Having created this mesh, the software uses the polygon mesh as a control cage; each polygon is

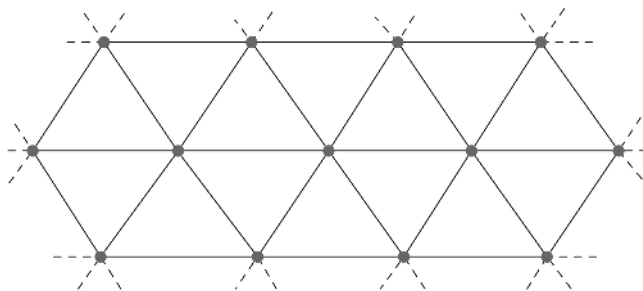


Figure 5.2 A 'standard' triangular mesh

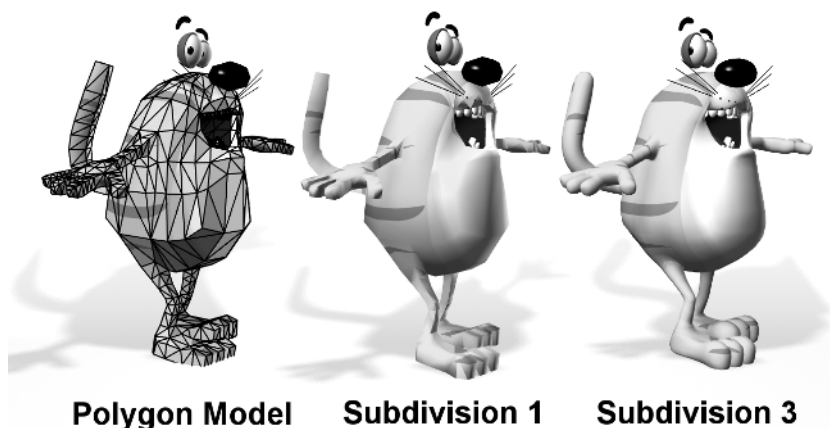


Figure 5.3 *Using subdivision*

converted into four polygons for each subdivision. With three or four divisions you have a smooth shape. Take a look at the first of the strip of images in Figure 5.3. Look at the point at the top of the cat's right shoulder. Notice that eight lines meet at this point, while the point immediately above it has just six lines meeting. This is possible for a polygon cage, indeed any number of lines can meet at a point, but a NURBS cage has to have the same number of lines meeting for every point.

To show you how the cat model was developed, we will look at the Lightwave modeler. As a polygon modeler, Lightwave is often considered to be the best there is and after using it you will quickly see why. The tools you use are very simple and easy to use. For the cat model we first start with a basic ball (see Figure 5.4).

All polygon modelers offer a ball creator. In Lightwave you can specify how many segments there are around the ball and how many slices. Having created the ball, you can then pull certain polygons around to create the inside of the mouth. If you look closely at the side view you can see that what used to be the cap is now inside and facing in the opposite direction. As you create a polygon model you will constantly be adding and modifying the geometry. In Lightwave a useful tool is the 'Knife', which allows you to cut through a model and so create more geometry. Another useful technique is to 'Smooth Shift' a polygon. Notice in Figure 5.4 that the top right view shows two sticks pointing out of two polygons. By highlighting the polygons in this way and selecting the Smooth Shift tool we can add geometry. The polygon is moved in the direction of the lines and four new polygons are added to fill in the gap.

We can repeat this method several times to create the geometry that we will use for the arms and legs (see Figures 5.5 and 5.6).

Having added the geometry we then resize the bits to fit the design we are creating. Next we need to create the hands and feet, which we do again by using the Smooth Shift and Knife tools, to create more geometry. Again a little dragging of the points and we have a rather clumsy hand shape (see Figure 5.7). One of the features of subdivision surfaces in Lightwave is that they tend to be slimmer than the control cage. So we often need to create a control cage that looks a little clumsy so that the subdivided result looks good.

Finally we can add teeth and other features.

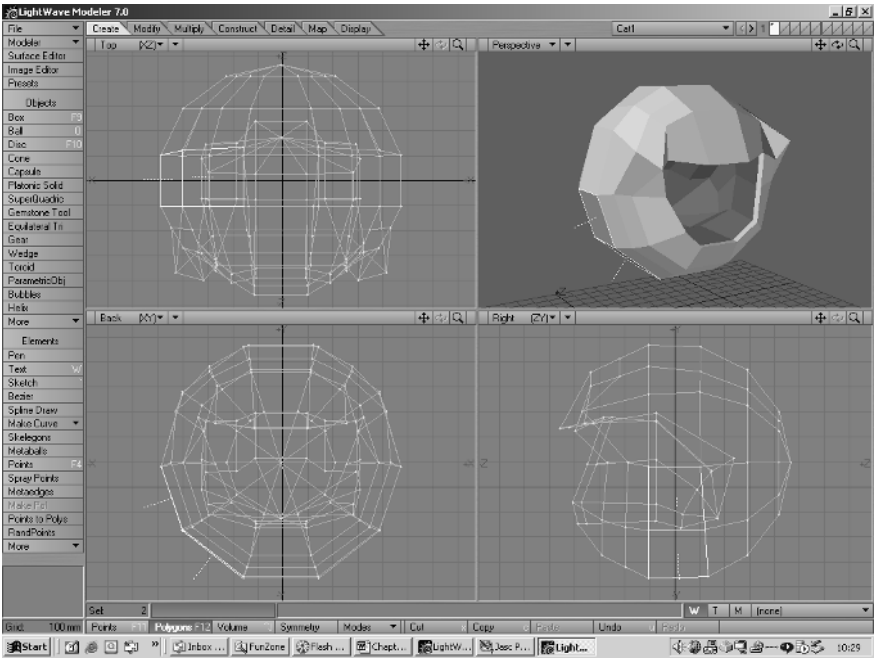


Figure 5.4 First step in creating the cat model

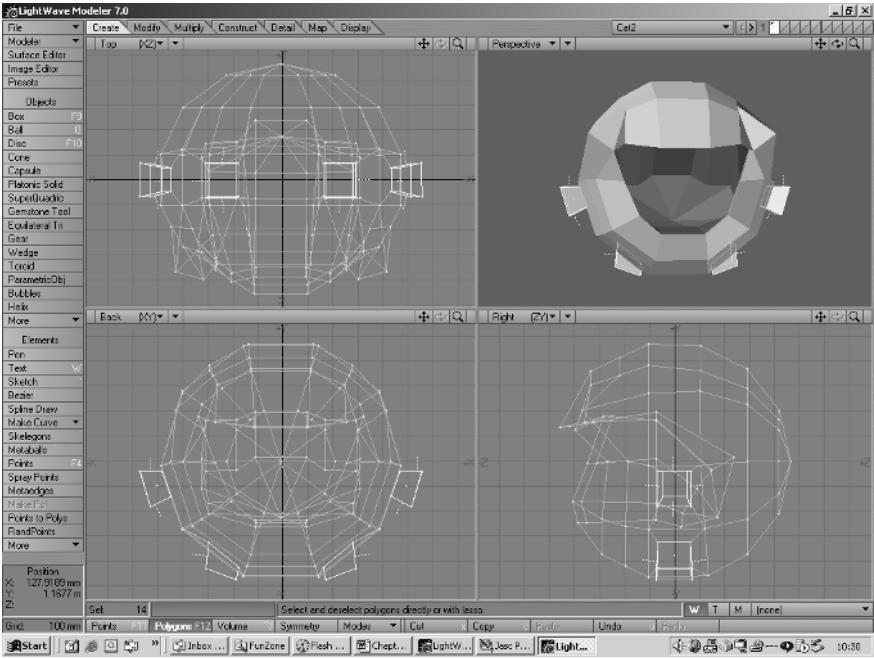


Figure 5.5 Adding the arms and legs

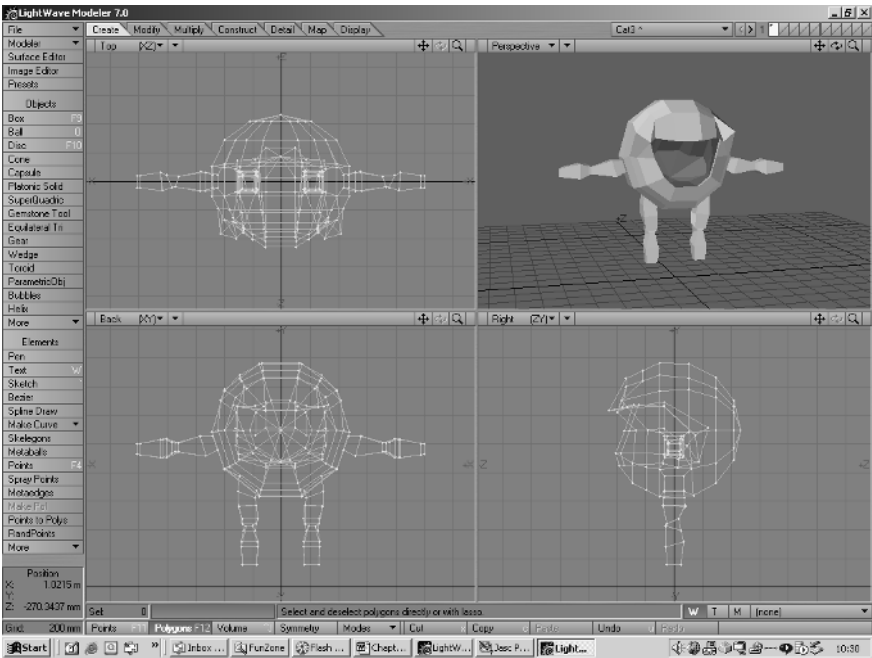


Figure 5.6 The arms and legs in place

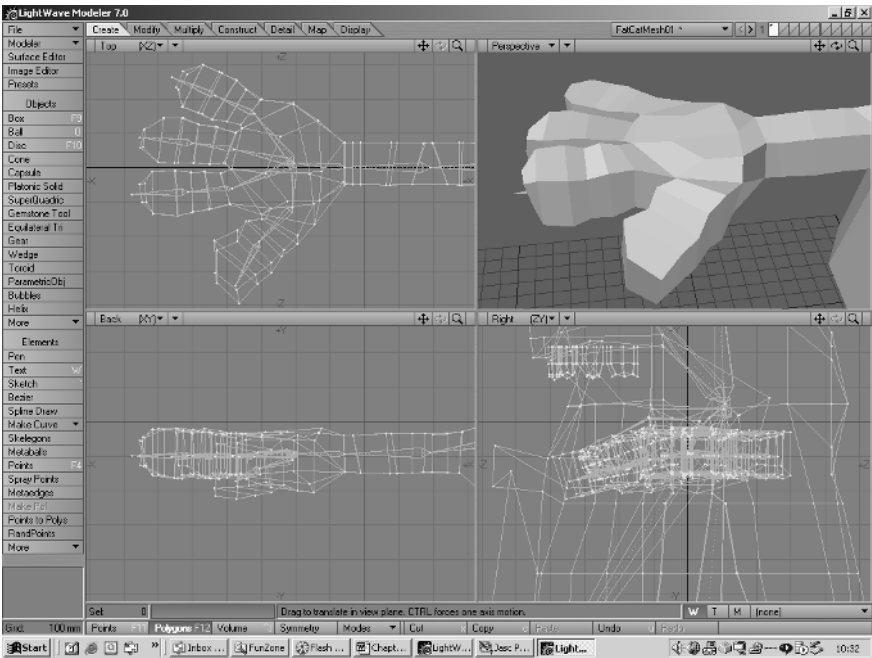


Figure 5.7 Creating a hand

When the geometry is complete it is time to set the coloring and surface details. For the cat we use a bitmap that is wrapped around the cat by the rendering software to create the stripes. This is called texture mapping. The texture map was created in Flash and then exported as a bmp file (see Figure 5.8).

When we have finished, we have a model that is starting to look like a very strange cat character (see Figure 5.9).

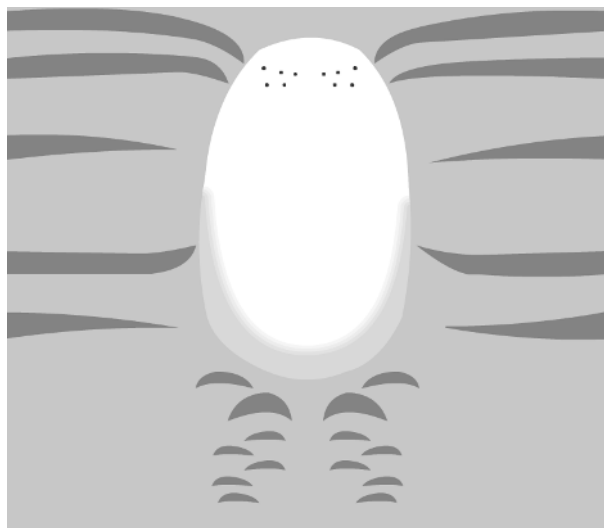


Figure 5.8 *The texture map that is used for the cat's body*

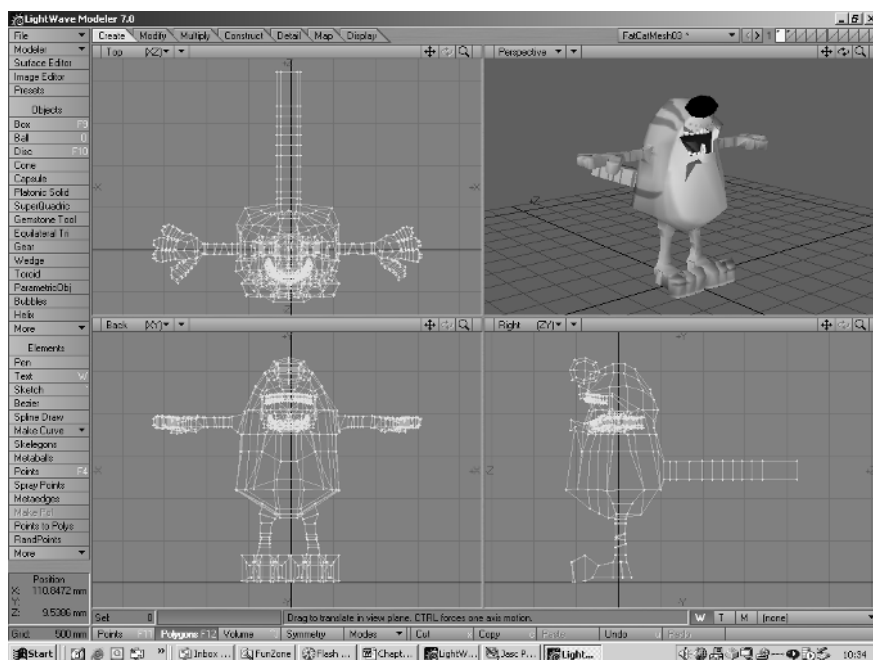


Figure 5.9 *The main body for the cat model*

Now we need to set this up so that we can animate the character. Most computer animation programs that can animate a character use some kind of 'bone' system. Lightwave is no exception. You can create the bones using the modeling tools. The purpose of a bone is to modify the position of the points in a mesh. Lightwave can decide which points a bone modifies on the basis of location. A point will be controlled more by nearby bones than by distant bones. This method works for some meshes but can cause errors in character animation, as limbs are so close together that the left thigh bone may influence the right thigh bone. Another technique is to create a definite relationship between a bone and certain points in the mesh. To set up a character to use bones in this way you create a vertex map, which is simply a list of vertices with each vertex having a specified weight. A weight of 100% means total control over the vertex while a weight of 50% means partial control. In this way you can control exactly which bone adjusts which points and errors are eliminated.

Animating the character

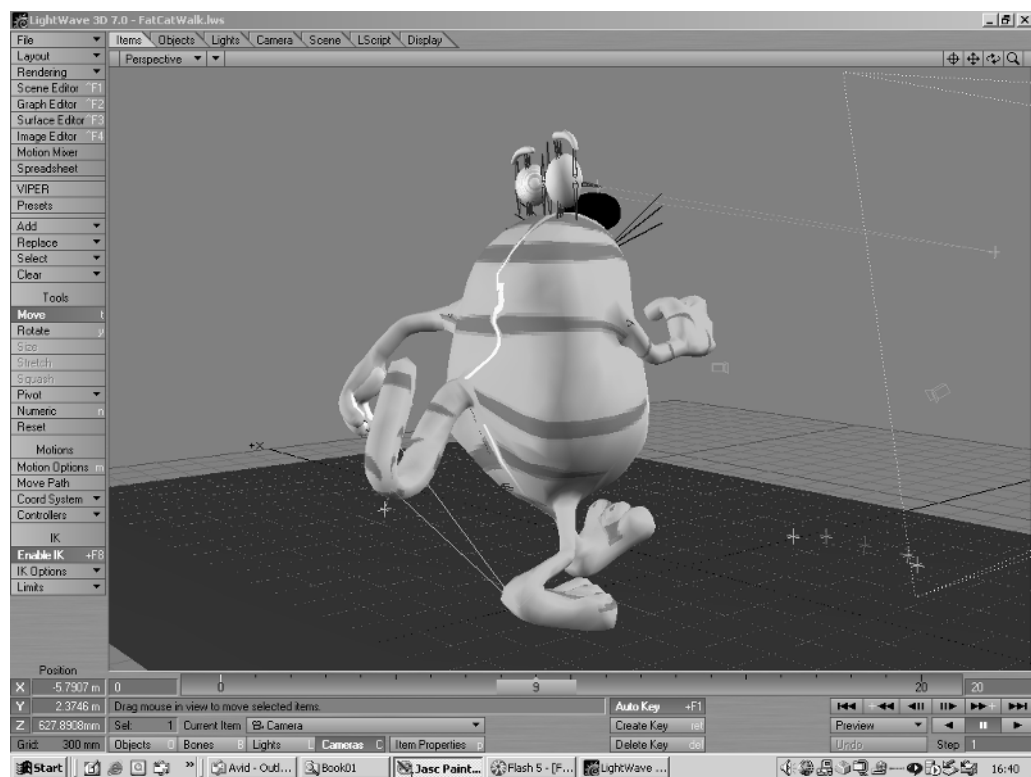


Figure 5.10 *Animating the cat character*

Once the character is set up to use bones you have control over each part of the mesh. Another useful feature of most computer animation programs is the use of inverse kinematics (IK). This is just a fancy name that means you can move the body but the feet will stay put. When animating a character, IK is a very useful feature and well worth setting up. You create a null object (an object that is used for animating but contains no geometry so it is invisible to the camera), which will become the target for the feet. Using IK, the feet will move to the target and can be made to rotate towards the target. The cat character uses IK on the feet, tail and eyes. Once set up correctly you can move and rock the body and the feet stay anchored, while the eye target ensures that the eyes stay locked in a direction even as the body moves.

Animating the character uses the principles that have been outlined in the previous chapter. For a walk you first put in the extremes, then the passing positions and finally the up and down positions (see Figure 5.11).

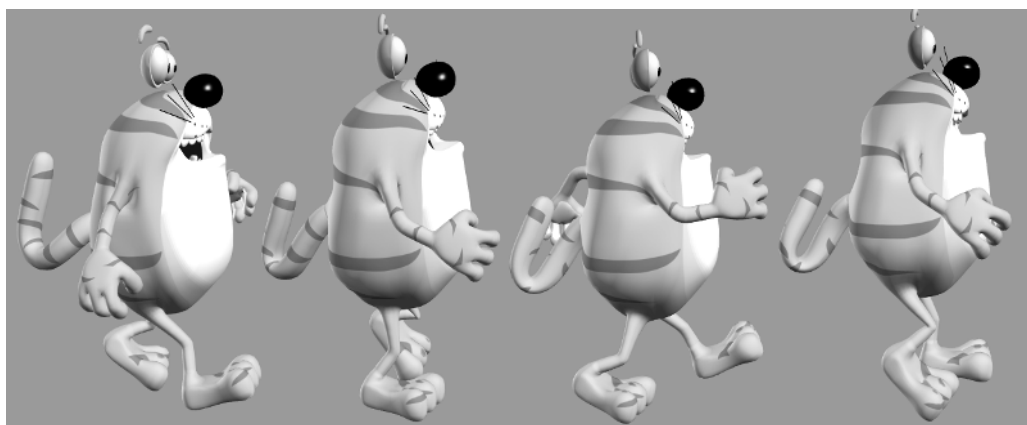


Figure 5.11 *The main keys in the cat's walk*

As you become experienced using a CGI program you will be able to create a short animation for a Flash game sprite very quickly.

Importing the animation into Director

The technique for importing the frames into Director is the same as that used in Chapter 2. Render the images as a 32-bit png file, to ensure that the alpha channel is included. Then import them into Director, use 'Modify/Cast to Time' to get them on the 'Stage' and then 'Publish' the movie to get it into the web-friendly compressed shockwave form (see Figure 5.12). You can find the images, a Director project file and a web page in the 'Examples/Chapter05' folder on the CD.



Figure 5.12 *The result published for the web*

Summary

In this chapter we looked at how an animator can use a computer animation program to create animation that can be imported into Director. In many ways computer animation programs represent the best solution for creating game assets for Director; they are useful for prerendered sprite games and they are essential when creating real-time 3D games.

Section 2

Scripting

Having developed the animation we now look at how to add interactivity to this art using Lingo or JavaScript.

This page intentionally left blank

6 So what is a variable?

After the first blistering introduction to Lingo in Chapter 1, we now take a more leisurely stroll through the basics of programming, starting with variables. The most fundamental aspect of programming is the ability to store and retrieve data. In essence all computer programming involves the manipulation of data. If your game uses a moving animated character running, jumping and shooting, you will need to store the position, size and action of the character as the game develops. This information is not static; it varies as the user plays the game. In this chapter we will look at how a variable can be used to store lots of different types of information including the data you would need for your character. We will look at how to set a variable's value and how to get the value stored in a variable. We will look at using chunks of variables in a structure called a *list*. Along the way we will examine the options using very simple Director movies.

What is Lingo?

Lingo is a list of instructions written in a very specific way, which is converted into a series of codes. When the Shockwave Player reads the codes, it performs the instruction in an unambiguous way. For example, if it found the instruction

```
x = 23
```

then it would check to see if it had a slot in memory called *x*, and if it had then it would set the value in this slot to 23; if it had no slot then it would create one and then set it to 23. You can add Lingo to a movie in several different ways; behaviors, movie scripts, parent scripts and scripts attached to cast members. We will look first at adding a score behavior script.

What is JavaScript?

Starting with Director MX 2004 a programmer can choose to use either Lingo or JavaScript to write code for Director. JavaScript uses a syntax that is familiar to C, C++, Java or Flash programmers. Just like Lingo it is a list of instructions written in a very specific way, which is converted into a series of codes; when the Shockwave Player reads the codes, it performs the instruction in an unambiguous way. In this and the next few chapters all the examples will be presented in both Lingo and JavaScript syntax. If you are totally new to Director then I would suggest using the JavaScript option because it will allow you to become familiar with code structures that you will find in other development packages.

Adding a score behavior script

Start a new movie; open the 'Score' window if it is not already open, using 'Window/Score'. Find the script row indicated by a small icon that looks like a page of text. Double click to the right of

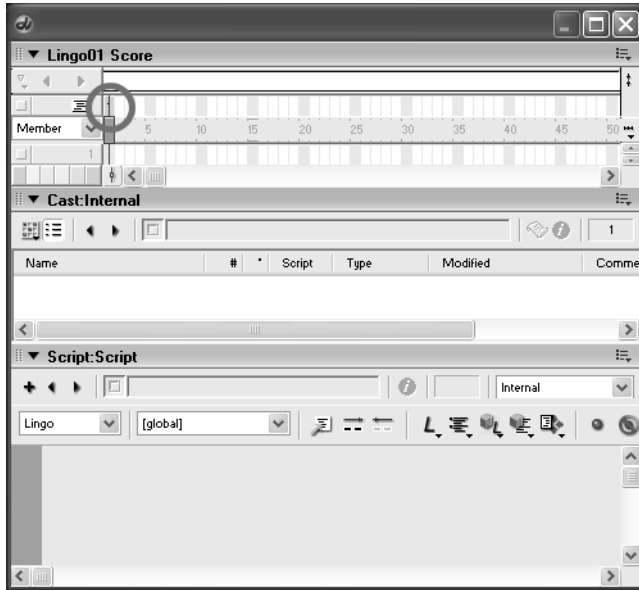


Figure 6.1 *Creating a score behavior script*

this in the slot at frame 1. You will see a ‘Script’ window open. Figure 6.1 shows where to click and shows the ‘Script’ and ‘Cast’ windows.

You will find that the script is created and populated with the following code:

```
1 on exitFrame me
2
3 end
```

Listing 6.1

Click on line 2 and enter this code:

```
x = x + 1
put x
```

You should have

```
1 on exitFrame me
2   x = x + 1
3   put x
4 end
```

Listing 6.2

This little snippet of code will be executed every time the playback head moves out of the frame. Because this movie has a single frame, Director will loop back over the frame continuously. So this code will be executed as many times per second as the movie frame rate is set to. So what is going to happen? The variable ‘x’ is going to be set to its own value plus 1 and then the result is going to

be displayed in the 'Message' window. The command 'put' is used to show details in the Message window.

You will also notice that the script is added to the 'Cast' window (see Figure 6.2).

OK, let's run the code.

Notice from Figure 6.3 that the value 1 is displayed repeatedly in the 'Message' window. Why? The answer lies in the persistence of the variable. We will look in more detail at this later but essentially 'x'

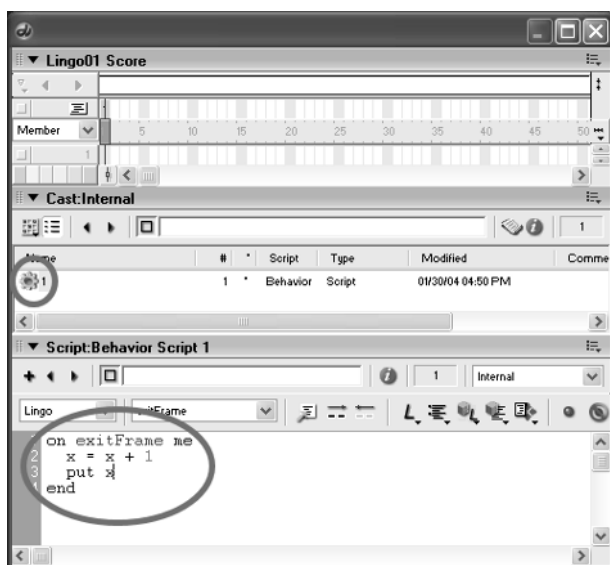


Figure 6.2 Entering Lingo in the exitFrame handler

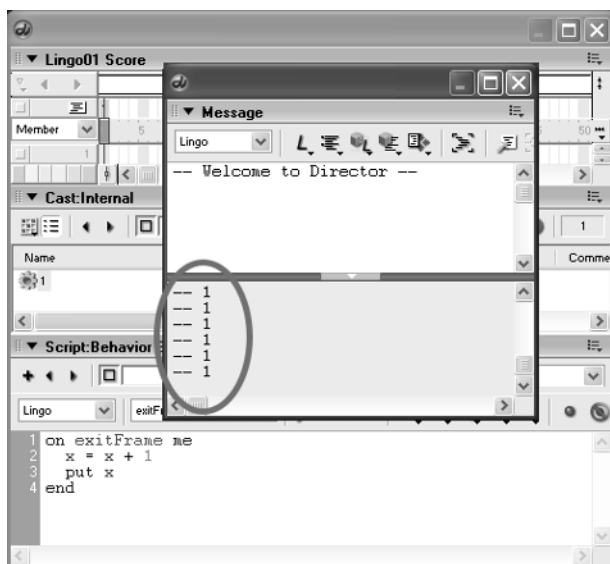


Figure 6.3 Viewing the results in the 'Message' window

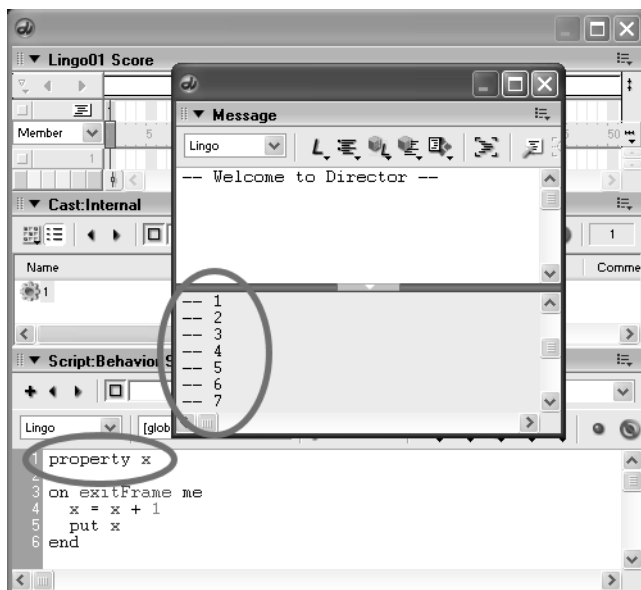


Figure 6.4 Making *x* a property

is recreated every time the code runs; because *x* does not have a value, the value is set to zero plus 1 or simply 1! We want *x* to persist from one execution of the code to the next. To facilitate this we make *x* a property of the script.

Just add the line of code shown in Figure 6.4 and run the movie. Now *x* is incremented every frame.

Using JavaScript

Notice that at the top left of the 'Script' window there is a drop-down box, which by default displays the word 'Lingo'. To use JavaScript, simply click the down arrow and change scripting from Lingo to 'JavaScript'. In this simple instance the code would then be

```
1 var x = 0;
2
3 function exitFrame(){
4   x = x + 1;
5   trace(x);
6 }
```

Listing 6.3

JavaScript functions do not use the passed parameter 'me'. All the properties available using me are available to the property 'this'. For example, the sprite channel can be retrieved in JavaScript using 'this.spriteNum'.

Using JavaScript, script-level variables are declared using the ‘var’ key word. This equates to the ‘property’ key word in Lingo. Where you would use ‘on myMethod arg...end’ with Lingo, JavaScript uses the key word ‘function’. All parameters to a function must be contained between brackets and all statements of the function between curly braces. A semi-colon follows each statement. The parallel to ‘put’ in Lingo is ‘trace’ in JavaScript (see box).

Lingo	JavaScript
property pProp	var pProp;
on myMethod arg put "Called myMethod with" & arg end	function myMethod(arg){ trace("Called myMethod with " + arg; }

What is a variable?

There are lots of ways to visualize a variable if the concept seems strange. One suggestion is to think of a telephone book. If you have a friend called Jim and his telephone number is 01234 567890 then you would put the number 01234 567890 under the name Jim in your book. Whenever you want to know what Jim’s telephone number is you look at the name Jim in the book and find out the number is 01234 567890. But suppose that Jim changes house and gets a new number. Perhaps the new number is now 09876 543210. You simply cross out the old number and put the new one in its place. Using the name Jim you can check the number; this is how you use a variable in a computer program.

To create a variable in Director you simply use it for the first time:

```
bigfeet = 14
```

This would create a variable called ‘bigfeet’ and set its current value to 14. Because you can add code in different places this is both very convenient and in some ways confusing, especially when you are writing your first programs.

The scope of a variable

A variable has something called *scope*. If you have read through Chapter 1, then you will realize that Director can use sprite ‘Behaviors’. Another type of script is a ‘Movie Script’. Take a look at ‘Examples/Chapter06/Lingo03.dir’, where we have added to the earlier example a second script. We used the ‘Script’ window, simply pressing the ‘+’ button. By default an empty Movie Script is created. We add the following code to the script; notice that all further listings in this chapter will be given for both a Lingo and a JavaScript version. You can only use one of these at a time, not both. But a movie can use both Lingo and JavaScript. For example, you may use a Lingo Movie Script and a JavaScript Behavior in the same movie.


```
--Lingo
1  global gX
2
3  on startMovie
4    gX = 123
5  end

//JavaScript
1  function startMovie(){
2    _global.x = 52;
3  }
```

Listing 6.4

The handler 'startMovie' occurs once for a movie, at the start! Movies can also have handlers for when they end and several other key events. We will look into this later.

By using the term 'global' when declaring the variable 'gX' we allow the variable to be accessed from other scripts. JavaScript uses the '_global' object to pass global variables around. Use the 'Score' window to locate the frame script found on frame 1 of the script channel. Then alter its contents using the listing below:

```
--Lingo
1  property x
2  global gX
3
4  on exitFrame me
5    x = x + 1
6    gX = gX + 1
7    put x && gX
8  end

//JavaScript
1  var x = 0;
2
3  function exitFrame(){
4    x++;
5    _global.x++;
6    trace(x + " " + _global.x);
7  }
```

Listing 6.5

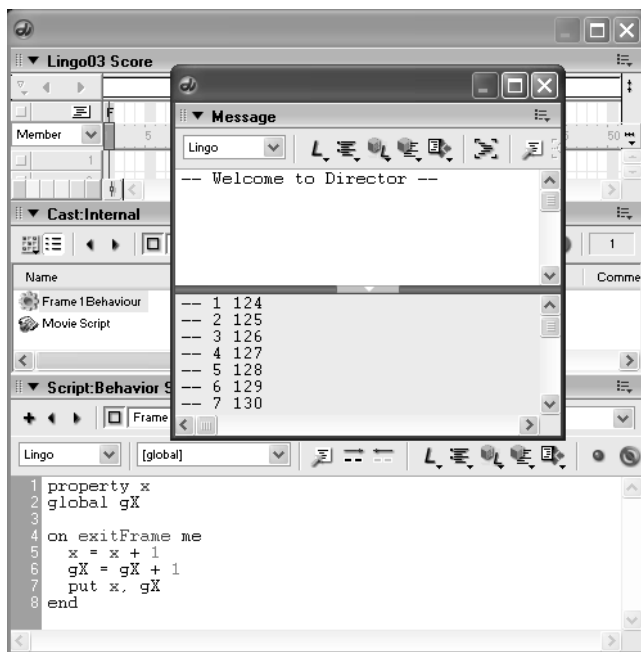


Figure 6.5 *Using global variables*

Notice that line 2, 'global gX', is also included in this script. This tells Lingo that any reference to gX refers to the global version not a local one. Without this gX would be assumed by Lingo to be local to the 'exitFrame' handler. The JavaScript version uses the '_global' object to achieve the same result. Also notice the '++' construction, which simply means add 1 to the variable name that precedes it.

A sprite behavior

The type of script you are likely to use most is a sprite behavior. Open 'Examples/Chapter06/Lingo04.dir' or 'Examples/Chapter06/JavaScript04.dir'. This movie includes a sprite; just a single image of the cat from the previous chapter is used. This is placed in the middle of the 'Stage'. Right click on the sprite on the 'Score' and select 'Script...' or right click on the cat on the Stage and select 'Script...'. The code used is shown in Listing 6.6.

```
--Lingo
1 global gX
2
3 on enterFrame me
4   spr = sprite(me.spriteNum)
5   if (_key.keypressed(123)) then
6     spr.locH = spr.locH - 1
7   else if (_key.keypressed(124)) then
8     spr.locH = spr.locH + 1
```

```

9   end if
10  gX = spr.locH
11 end

--JavaScript
1 function enterFrame(){
2   var spr = sprite(this.spriteNum);
3   if (_key.keyPressed(123)){
4     spr.locH--;
5   }else if (_key.keyPressed(124)){
6     spr.locH++;
7   }
8   _global.gX = spr.locH
9 }

```

Listing 6.6**Lingo**

Line 1 uses the global declaration for the variable ‘gX’, so the script knows it is dealing with a variable accessible from anywhere in the movie. A sprite behavior can include many handlers, one of the most common being ‘enterFrame’. An enterFrame handler can be followed by a parameter, in this case the key word ‘me’. The me variable contains information about the object that the script is attached to. The one we want is the actual sprite we see on screen. To access this we use the code in line 4. This sets a local variable ‘spr’ to the sprite on the ‘Stage’ in the channel defined by ‘me.spriteNum’. We then use a control structure that we will see lots more of:

```

if (something is true) then
  --execute some code
end if

```

The thing we want to test is whether the left or right arrow keys are pressed. The code for this is

```
_key.keypressed(123)
```

for the left arrow key and

```
_key.keypressed(124)
```

for the right. If the key is pressed then the return value will be true.

Using the information about key presses we update the sprite’s horizontal location using lines 6 or 8. Finally we set the global variable ‘gX’ to the current value of the sprite’s horizontal location.

JavaScript

A sprite behavior can include many handlers, one of the most common being ‘enterFrame’. An enterFrame handler can be followed by a parameter, in this case the key word ‘me’. The me variable contains information about the object that the script is attached to. The one we want is the actual sprite we see on screen. To access this we use the code in line 2. This sets a local variable ‘spr’

to the sprite on the Stage in the channel defined by 'me.spriteNum'. We then use a control structure that we will see lots more of

```
if (something is true){
    //execute some code
}
```

The thing we want to test is whether the left or right arrow keys are pressed. The code for this is

```
_key.keyPressed(123)
```

for the left arrow key and

```
_key.keyPressed(124)
```

for the right. If the key is pressed then the return value will be true. Note that JavaScript uses the '_key' object to access this function and that the function name is case sensitive, using a capital 'P'.

Using the information about key presses we update the sprite's horizontal location using lines 4 or 6. '--' simply means decrease the variable by 1 and '++' increase the variable by 1. Finally we set the global variable 'gX' to the current value of the sprite's horizontal location, using the '_global' object.

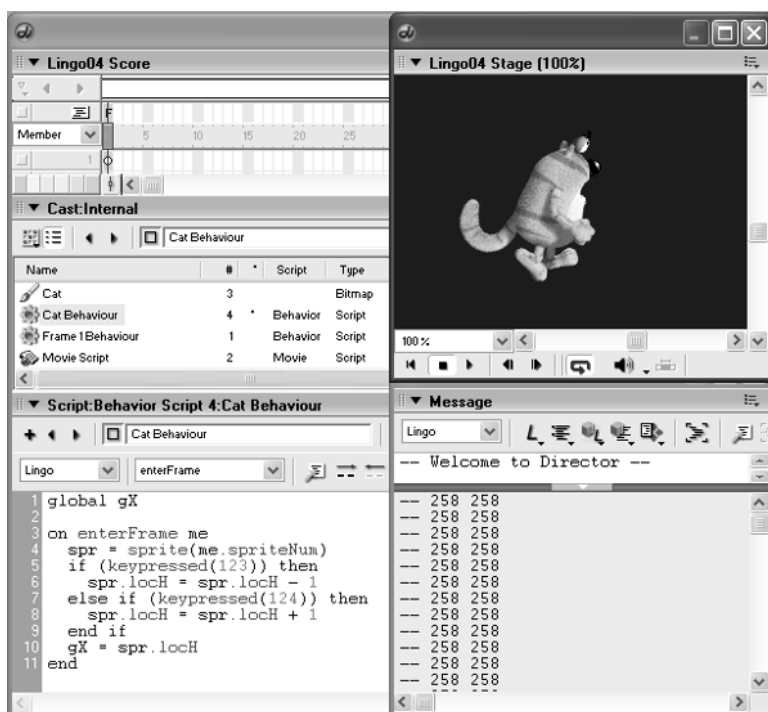


Figure 6.6 Adding a sprite behavior

Creating and using functions

You've seen how you can place code in different places in a movie. Each script you have examined has used some kind of handler. When a certain event occurs, the start of a movie or the entry or exit from a frame then the code between 'on ... end' is executed. Within the same script you can create your own blocks of code, called functions. The purpose of a function is to package the execution of your script and to make the writing and maintenance easier. Take a look at 'Examples/Chapter06/Lingo05.dir' or 'Examples/Chapter06/JavaScript05.dir'.

```
--Lingo
1 on exitFrame me
2   myFunction()
3 end
4
5 on myFunction()
6   put "Hello from myFunction"
7 end

//JavaScript
1 function exitFrame(){
2   myFunction();
3 }
4
5 function myFunction(){
6   trace ("Hello from myFunction");
7 }
```

Listing 6.7

Lingo

The movie just contains a frame script on frame 1. The Lingo in the script is shown in Listing 6.7. Notice that there are now two code blocks, an 'exitFrame' handler and a 'myFunction' function. To create a function you just use 'on xxx ... end' where 'xxx' is a useful name that you can use for the function. The name must be unique and cannot be a Lingo key word. To call the function just use the name you have chosen. In the example shown the function myFunction is called from the exitFrame handler as it loops around.

JavaScript

The movie just contains a frame script on frame 1. The JavaScript in the script is shown in Listing 6.7. Notice that there are now two code blocks, an 'exitFrame' function and a 'myFunction' function. To create a function you just use 'function xxxx(){ ... }' where 'xxx' is a useful name that you can use for the function. The name must be unique and cannot be a JavaScript key word. To call the function just use the name you have chosen followed by parentheses. In the example shown the function myFunction is called from the exitFrame function as it loops around.

A function can be passed information in the form of parameters or arguments. 'Examples/Chapter06/Lingo06.dir' shows an example.

```
--Lingo
1 property x
2
3 on exitFrame me
4   x = x + 1
5   showTheNumber x
6 end
7
8 on showTheNumber n
9   put "The number is " & n
10 end

//JavaScript
1 var x = 0;
2
3 function exitFrame(){
4   x++;
5   showTheNumber(x);
6 }
7
8 function showTheNumber(n){
9   trace( "The number is " + n );
10 }
```

Listing 6.8

Here we use the function 'showTheNumber'; this time the function name is followed by a variable. In the 'exitFrame' handler we pass the variable 'x'. In the function showTheNumber, whatever is passed is referred to within the function as n. You may find this very confusing; one minute it is x the next it is n, so which is it? The truth is we could use

```
showTheNumber 6 * 7
```

In this case Lingo would calculate the value of $6 * 7$, then within the function code this value would be assigned to the variable n. The code at line 9 would therefore be

```
put "The number is " & 42
```

The effect of this code is to create a sentence that combines the words 'The number is' with the value 42. Think of the parameter n as a slot; when we call the function we put something in that slot, then whatever is in the slot is what the function uses. The only significant difference between the Lingo and JavaScript syntax is the use of the initialization of the variable x in line 1. It is essential to initialize JavaScript variables to a known value before using them, to avoid spurious results.

We can use more than one parameter and we can even return a value back to the caller. Take a look at 'Examples/Chapter06/Lingo07.dir' or 'Examples/Chapter06/JavaScript07.dir'.

```

--Lingo
1 property x
2
3 on exitFrame me
4   x = x + 1
5   put addNumbers(x, 23)
6 end
7
8 on addNumbers a, b
9   return a + b
10 end

//JavaScript
1 var x = 0;
2
3 function exitFrame(){
4   x++;
5   trace( addNumbers(x, 23) );
6 }
7
8 function addNumbers( a, b ){
9   return (a + b);
10 }

```

Listing 6.9

In this example we pass two values into the function. When multiple values are used we separate them with a comma. Then in the function we add these values together (line 9) and use the key word 'return' to send this value back to whoever called the function. In this case it was at line 5, so the sum is then shown in the 'Message' window. Whenever a function returns a value the calling code must place parentheses around the parameters; when using JavaScript parentheses are always required.

`addNumbers(x, 23)` not `addNumbers x, 23`

As you become more experienced at writing code you will begin to use functions, which help keep your code simple to understand. There is a perceived wisdom that if a block of code exceeds what can be seen on a single screen then you need to split up your code more. There is a good deal of truth in this maxim and in later chapters you will learn more about how good structure helps make good games.

Some different types of variable

Although a variable can be thought of as a pigeonhole in which you store some information, this information may be a number, a word, a sentence or even multiple values via a list. In some programming languages, you, the developer, must decide what kind of variable you are going to use

Table 6.1 *Variable types*

Variable	Value
Sword	false
HorseHeight	43
HorsePower	23
KnightName	Lancelot
BattleLocation	The Forest of Dean
KnightAge	23.78

Table 6.2 *Variable types*

Variable	Value
var1	false
var2	43
var3	23
var4	Lancelot
var5	The Forest of Dean
var6	23.78

and once set it must stay this way for the duration of the program. In Director you need not declare a variable, it is created the first time it is used and can be changed in its use from a number to a sentence if necessary. Table 6.1 shows some possible variables and their values. Table 6.2 shows an alternative list.

Although the values are the same the names are not. As a developer you can choose the name that you find most useful. When people speak of a programming language, that is what you need to learn; you need to understand what will happen when you write a section of code, but you don't need to make your life any harder than it needs to be and one of the great benefits of using a programming language is that you usually have the convenience of naming your variables as you think best. Over the years lots of developers have thought about the naming of variables and it is always best if your naming makes sense; try not to abbreviate the names; always use 'carspeed' rather than 'cs'. The different names make no difference to the performance of your program, but they do make a great deal of difference to the readability of your code. You may think that you will always understand the way you have written a program, but you won't. Even a week later a complex section of code will seem mystifying. Remember the golden rule 'keep it simple, stupid' (KISS).

How decimal values differ from integers

The first type of variable we will consider in any depth is a number. Numbers come in literally an infinite amount of values. Unfortunately computers, although fast and capable of storing very large and very small values, cannot store an infinite number of values. The way that developers have got around the limitations inherent in computer storage is to use two distinct types of number, *integers* and *floats*. The best thing about integers is that they are exact.

$$2 * 12 = 24.$$

The * symbol, not the \times symbol, is used to represent multiplication on a computer. But the range of values that can be stored in an integer is limited, not very limited as you can use integers to store values into the millions, but not into the trillions. Similarly, integers are not very useful if your numbers all range between zero and 1. For the very big and the very small you can use floats. There is a potential problem, however, as floats are not exact. They may seem exact, but they are not. Suppose you multiply 0.01 by itself lots of times. The number will get smaller:

```
0 = 0.0001
1 = 0.00001 or 1e-6
2 = 0.0000001 or 1e-8
3 = 0.000000001 or 1e-10
4 = 0.00000000001 or 1e-12
...
152 = 1e-308
153 = 9.99999999999998e-311
154 = 9.99999999998466e-313
155 = 9.99999999963881e-315
156 = 9.99999983659715e-317
157 = 9.99998748495601e-319
158 = 9.99988867182684e-321
159 = 9.88131291682494e-323
160 = 0
```

After the 160th iteration, the value becomes zero. This may seem unlikely, but a computer would whizz through the calculation in a blink of an eye. Once the value has diminished to zero the quantity has disappeared. If the intention was to use the result to multiply another variable then this too will become zero. Floating point errors like this are another source of difficult-to-locate bugs in a program. Always remember that unless you are working with integers within a sensible range, all other values are approximate. You may test two floating point values to see if they are the same using

```
if (number1 = number2) then
  --Do something
end if
```

Although this would work fine for integers, don't use it for floats; instead use

```
if ((number1 - number2)*(number1 - number2)) < 0.0000001) then
  --Do something
end if
```

By subtracting the second number from the first you get the difference between two numbers. This could be positive or negative, but by multiplying the difference by itself you guarantee it will be positive, as any number multiplied by itself is positive. Now you can test to see whether this is a small number. The choice of 0.0000001 is arbitrary and could be smaller if necessary; the

important thing is that you use a difference test and not an equality test. Another way to guarantee a positive value is to use

```
if (abs(number1 - number2)) then
  --Do something
end if
```

Here the operation ‘abs’ is used, which is short for absolute value. You may prefer this method.

If you are totally new to programming then you will probably be in something of a daze at this time. This feeling is natural. We speak of programming *languages*. You may feel like a stranger in a strange land, where nothing is what it seems and nobody speaks your language. As you start to pick up a few words, you can at least order a coffee and buy a loaf of bread. In this chapter we are rather forced to introduce a few words and a little punctuation. Please bear with us and in the end you will no longer be a stranger.

Using strings

When we use a series of letters then we create what is known in computer jargon as a string. ‘Examples/Chapter06/Lingo08.dir’ or ‘Examples/Chapter06/JavaScript08.dir’ illustrate an example. For a change we use the ‘beginSprite’ handler, which occurs just once as the sprite starts. Listing 6.10 shows that on line 2 a string is defined.

Lingo

The remaining lines access information about the string and subsets of the string. Line 3 simply displays the string in full. A Lingo string is divided up into words by the space character. You can access the words at any time. Line 4 uses a property of the words, ‘count’, to find out how many words are in the sentence. You can access any of the individual words using

```
word[x]
```

where ‘x’ is the number of the word you want to access. A word is simply a sequence of characters; these can be accessed using

```
char[x..y]
```

where x is the start of the character subset and y the end. Listing 6.10 is followed by the output from running it.

JavaScript

The lines following line 2 access information about the string and subsets of the string. Lingo and JavaScript access strings entirely differently. To allow you to easily access the words in a string you need to split the string into separate words using the code in line 3. The ‘split’ method takes a single parameter that is the character used to split up the string. Whenever it finds one of these characters it adds what preceded it to a list of values that JavaScript refers to as an array. In line 3 we split the string whenever we find a space character; another common character used in splitting is a comma. Line 4 simply displays the string in full.

The variable 'myWords' now contains an array of separated words we can access easily. Line 5 uses a property of an array, 'length', to find out how many words are in the sentence. You can access any of the individual words using

```
myWord[x];
```

where 'x' is the number of the word you want to access. Remember that indices in JavaScript always start at zero. Each value in the array is a string. JavaScript has several methods for accessing the individual characters. Lines 7 and 8 are a single statement. JavaScript allows the programmer to enter a single statement across several lines; the semi-colon defines the end of a statement. In Line 8 we use the method 'charAt'

```
charAt(x);
```

where x is the index of the character to access.

Lines 9 and 10 include another JavaScript string method, 'substring'. This takes two parameters, the first is the first character to extract and the second is the first character to skip. The return value of this method is a new string that includes all the characters between the two values; it includes the first character index but excludes the last.

Listing 6.10 is followed by the output from running it.

```
--Lingo
1 on beginSprite
2   myString = "This is a sentence using seven words"
3   put "The sentence is: " & myString
4   put "The number of words in the sentence is" & ↵
       myString.words.count
5   put "The third word is" & myString.word[3]
6   put "The second letter of the fifth word is " & ↵
       myString.word[5].char[2..2]
7   put "The middle of the fourth word is" & ↵
       myString.word[4].char[4..6]
8 end

--JavaScript
1 function beginSprite(){
2   myString = "This is a sentence using seven words";
3   myWords = myString.split(" ");
4   trace("The sentence is:" + myString);
5   trace ("The number of words in the sentence is " + ↵
       myWords.length);
6   trace ("The third word is" + myWords[2]);
7   trace ("The second letter of the fifth word is" +
8       myWords[4].charAt(1));
```

```

9   trace ("The middle of the fourth word is" +
10         myWords[3].substring(3, 5));
11 }

```

Listing 6.10

```

--'The sentence is: This is a sentence using seven words'
--'The number of words in the sentence is 7'
--'The third word is a'
--'The second letter of the fifth word is s'
--'The middle of the fourth word is ten'

```

Table 6.3 *Using strings in Lingo*

String	This is a sentence using seven words						
word	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	This	is	a	sentence	using	seven	words
Word	sentence						
char	[1..1]	[2..2]	[4..6]				
	s	e	ten				

In later chapters we will look in more detail at using the various string slicing facilities available in Lingo and JavaScript.

Using lists or arrays

Another very useful thing when using Lingo is a list. JavaScript calls these arrays. To define a list using Lingo you use the square braces '[..]'. To define them using JavaScript you generate a new array. Line 2 of Listing 6.11 shows a list being initialized. The code for this can be found in 'Examples/Chapter06/Lingo09.dir'. You can use any type of items in a list or array, they do not have to be of the same type. Each item in a list or array is separated using a comma when they are defined. Optionally you can name each item in a 'PropertyList', not an array, using

```
#yourname:
```

To access a value in a list you use

```
myListName[n]
```

where 'n' is the index of the item in the list. If you have defined names for the items then you can access an item using

```
myListName.myItemName
```

An example helps clarify the concept:

```

--Lingo
1 on beginSprite
2   myList = [#word:"Hello", #value:32, #list:[1, 2, 3, 4]]

```

```

3  put "The length of myList is" & myList.count
4  put "The first item in the list is" & myList[1]
5  put "The first item can also be found this way" & myList.word
6  put "The value is" & myList.value
7  put "An embedded list" & myList.list
8  put "The third item in the embedded list is" & ↵
    myList.list[3]
9  end

--JavaScript
1  function beginSprite() {
2    myArray = new Array("Hello", 32, new Array(1, 2, 3, 4) );
3    trace( "The length of myList is" + myArray.length);
4    trace( "The first item in the list is" + myArray[0]);
5    trace( "The second value is" + myArray[1]);
6    trace( "The embedded list is" + myArray[2]);
7    trace( "The third item in the embedded list is" + ↵
        myArray[2][2]);
8  }

```

Listing 6.11

Lingo

A list has a 'count' property that returns the number of items; this is used in line 3. Lines 4–6 illustrate the two ways that you can access an item in a list, either by index number or name, if one has been used. A list can include another list. Lines 7 and 8 show this in operation; the name of an embedded list is entirely your own choice and does not have to be 'list' as in the example.

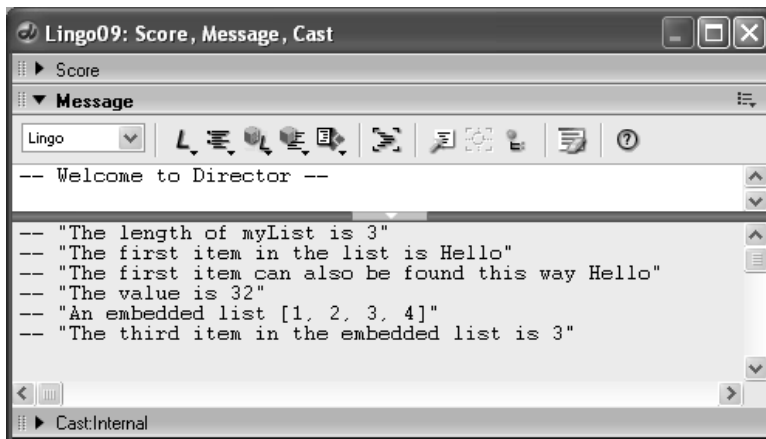


Figure 6.7 The output from 'Lingo09.dir'

JavaScript

An array has a 'length' property that returns the number of items; this is used in line 3. Lines 4 and 5 illustrate how you can access an item in an array. An array can include another array. Lines 6 and 7 show this in operation.

At first glance the naming facility may suggest that Lingo has some superior syntactic assets. Unfortunately this is not the case. JavaScript uses a much more sophisticated method, 'Classes', to allow name access to blocks of data. We will cover this in Chapter 9.

Summary

We have covered a great deal in this chapter, from creating a basic variable to much more complex string slicing. Along the way the structure you will use in many of your Director movies has been introduced. You have learnt how to add Lingo or JavaScript to a frame script, a movie script and a behavior. Along the way you have come across several elements in Lingo and JavaScript including the 'if ... then' structure. In the next chapter we will look at using this most useful of commands in greater detail.

7 In tip-top condition

Should I go to Barcelona at Easter? If I have the money, if there are flights available and if I can get a hotel, then I'll go. Life is full of decisions. Should I take that job? Should I join a gym? Should I make a cup of tea? Whenever you make a decision, you will consider a number of options and in the end you will either decide to do something or you will choose not to. As you embark on your programming career you may be thinking if this doesn't get any easier then I stick to producing artwork. In this chapter we look at making decisions in your programs using the marvelous 'if ... then' structure. An if ... then structure takes a logical argument that resolves to true or false; none of the difficult in-between concepts for a computer, it's either true or it's false and no half measures. Sometimes we need to combine several decisions before we come to our final conclusion. Multiple decisions are handled using Boolean logic, which is introduced in this chapter. We will illustrate some of these concepts using a keyboard-controlled walking bucket and some very attractive lights. So without further ado let us set sail on the sea of conditions.

The marvelous 'if ... then' structure

Because it is almost impossible to write any program without making a decision, you have already seen the 'if ... then' structure in action. The introductory chapter used the if ... then structure and so too did the previous chapter. You probably had a good idea what was going on, but you were not properly introduced. The marvelous if ... then structure can be used in one of three different ways. The simplest way is like this:

```
--Lingo
if (condition) then
  --Do something if condition is true
end if

//JavaScript
if (condition){
  //Do something if condition is true
}
```

The line starts using if ... then, and then in parentheses we put a condition. This can be anything that evaluates to true. You can even put the key word 'true' in there and then the if ... then structure always evaluates to true, so the code between 'then' and 'end if' or { and } is always executed. Later in this chapter we will look at the different conditions you can use.

The second variant of the 'if ... then' structure takes this form:

```
--Lingo
if (condition) then
```

```

    --Do something if condition is true
else
    --Do something if condition is false
end if

//JavaScript
if (condition){
    //Do something if condition is true
}else{
    //Do something if condition is false
}

```

This time we can not only execute code if the condition is true we can also execute different code if the condition is false.

The third and final form of the ‘if ... then’ structure takes the form:

```

--Lingo
if (condition1) then
    --Do something if condition1 is true
else if (condition2) then
    --Do something if condition2 is true
else
    --Do something if neither condition1 nor condition2 is true
end if

//JavaScript
if (condition1){
    //Do something if condition1 is true
}else if (condition2){
    //Do something if condition2 is true
}else{
    //Do something if neither condition1 nor condition2 is true
}

```

Here we have introduced ‘else if’. We can have as many else ifs as necessary for the logic in the program. The final else is optional and is only executed if all the previous ifs and else ifs in the block evaluate to false.

Tell me more about conditions

Director uses logical operators to decide whether a condition is true or false. The logical operators it supports are shown in Table 7.1.

All of these operators take a left operand and a right operand (in Table 7.1 described as ‘a’ and ‘b’ respectively). Either operand can be a variable or a fixed numeric value. Suppose you want to know when a character on screen is at a certain position (100,200), that is 100 pixels to the right and 200 pixels down from the upper left corner. You can test for an actual position using the following code snippet.

Table 7.1 Logical operators in Director

Operator	Description	Possible usage
a = b Lingo	Evaluates to true if a equals b	x = 23 Lingo
a == b JavaScript		x == 23 JavaScript
a > b	Evaluates to true if a is greater than b	x > 12
a < b	Evaluates to true if a is less than b	x < -22.3
a >= b	Evaluates to true if a is greater than or equal to b	x >= y
a <= b	Evaluates to true if a is less than or equal to b	x >= -y
a <> b Lingo	Evaluates to true if a is not equal to b	x <> 1 Lingo
a! = b JavaScript		x != 1 JavaScript

```
--Lingo
spr = sprite(me.spriteNum)
if (spr.locH = 100) then
  --Sprites locH (horizontal location) equals 100
  if (spr.locV = 200) then
    --Sprite at (100, 200)
    --Do something
  end if
end if

//JavaScript
spr = sprite(this.spriteNum);
if (spr.locH == 100){
  // Sprites locH (horizontal location) equals 100
  if (spr.locV == 200){
    //Sprite at (100, 200)
    //Do something
  }
}
```

Notice how in this example we nest one ‘if ... then’ structure inside another. You can do this as many times as you choose, but if the nesting is more than three you are encouraged to find another way to execute the code because nested if ... then structures are very prone to difficult-to-detect bugs; sometimes a condition is not met and the code goes screwy. Please note that Lingo and JavaScript have different syntax for the equality case. In Lingo you use ‘=’ to mean set something equal to something or test for equality. The circumstance of the code dictates the operation. In JavaScript ‘=’ always means assign a value to a variable and ‘==’ is used to test for equality. It is easy to get this wrong; when you do, the result of the operation is always true.

```
//JavaScript
var n = 12;
```

```

if (n = 10){
    //Will always execute leaving n equal to 10
}

if (n == 10){
    //Will only execute if n is equal to 10
}

```

Getting the distance between points on the screen

The distance between two points on the screen can be found using a principle that you learnt at school, Pythagoras' theorem.

Now you may be thinking how can Pythagoras help with this problem? The answer is that the relationship between the three sides of a right-angle triangle is one of the most useful devices you will find when creating graphical games. Let's start by reviewing the concept. If we have a triangle ABC where side AC has length x and side BC has length y then side AB must have a length which is the square root of the sum of the squares of the two other sides. Looking at Figure 7.1, this means that the area of the square attached to the side AC plus the area of the square attached to the side BC is equal to the area of the square attached to the side AB. Oh dear, this unfortunately sounds like a maths lesson. Fear not, you are not going to have to do any proofs or solve any complex algebra.

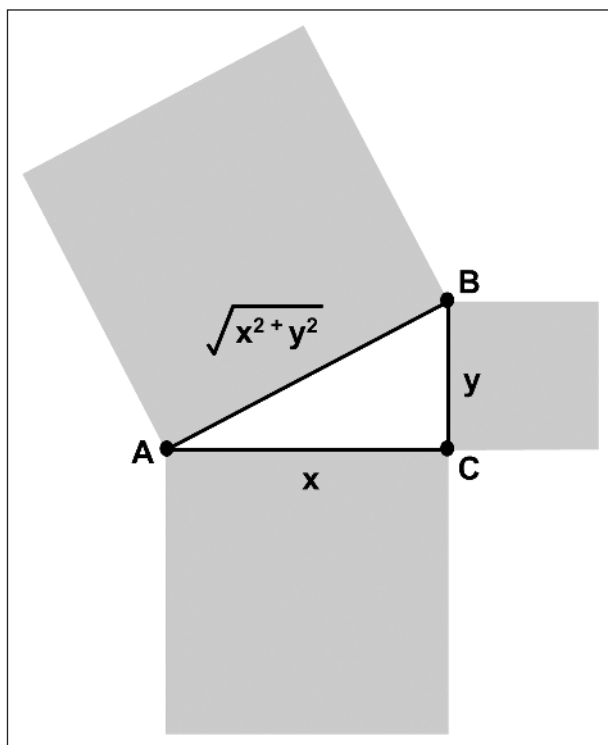


Figure 7.1 *That pesky Pythagoras*

No, we are just going to use this very useful fact to get the distance between two points on our screen. Suppose we have point A and point B. Then we know from our good friend Pythagoras that the distance from A to B is found by squaring the distance from AB in the x direction and squaring the distance from AB in the y direction. If we add these two results and find the square root of the sum then we have the distance from A to B. In Director code we can turn this into a useful little function.

```
--Lingo
on distanceBetween x1, y1, x2, y2
  x1 = x1 - x2
  y2 = y1 - y2
  return (sqrt(x * x + y * y))
end

//JavaScript
function distanceBetween( x1, y1, x2, y2){
  x = x1 - x2;
  y = y1 - y2;
  return (Math.sqrt(x * x + y * y));
}
```

In JavaScript access to the maths functions comes via the 'Math' object hence the last line of the JavaScript function.

If you understand the above then go to the top of the class; if it all seems a little hazy then let's take it a bit at a time. A function can be passed any number of parameters; they are the variables 'x1, y1, x2, y2'. If we have a point on the screen (100, 100) and another point (200, 200) then we can pass this information into this function by using

```
...
dist = distanceBetween(100, 100, 200, 200)
...
```

What happens when Director sees this code is that the player looks for a function, in the current scope, with the name 'distanceBetween'. If it does not find the function then Director will complain, showing the message box shown in Figure 7.2.

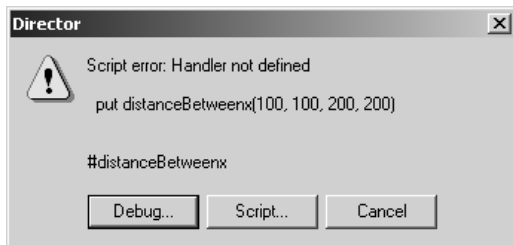


Figure 7.2 Director indicates a handler not found

If it does find the function then it runs the code in the function; because of the values we have used in this example

```
x1 = 100, y1 = 100, x2 = 200 and y2 = 200
```

The order they appear in the parentheses dictates the value assigned to each of the parameter variables in the function. If we change the order then the variables will be assigned different values, for example

```
...
dist = distanceBetween(100, 100, 200, 200);
...
```

assigns

```
x1 = 100, y1 = 100, x2 = 200 and y2 = 200
```

Within the function we use these values to get the distance between the points along each axis, using

```
x = x1 - x2;
y = y1 - y2;
```

Now if you are being extra smart you will realize that both *x* and *y* could take negative values if *x2* or *y2* are greater than *x1* and *y1* respectively. How do we handle this case? The fact is that in the function we make use of the square of these values and a square can never be negative. If we multiply a positive or negative number by itself then we get a positive value. For example,

```
-2 * -2 = 4 not -4
```

A function can return a value; to do this we use the key word `return`. In this example we also use the square root function, which takes a single parameter and returns the square root of this parameter. In this instance we are interested in the sum of the two squares.

Now we can make use of this function to test the distance between two points:

```
--Lingo
spr = sprite(me.spriteNum)
if (distanceBetween(100, 100, spr.locH, spr.locV)<2) then
  --Do something
end if

//JavaScript
spr = sprite(this.spriteNum);
if (distanceBetween(100, 100, spr.locH, spr.locV)<2){
  //Do something
}
```

Because the function returns a value we can use it as one side of a logical operator. In this instance we ‘do something’ if the distance between the sprite, *spr*, and the screen point (100, 100) is greater than 2. This test is much more robust and unlikely to do strange things. But a square root is quite a complex operation and in this instance not really necessary; we can use the square of the distance

between two points for the test. If we want the distance to be less than 2, then we want the squared distance to be less than 2 squared, or 4. We can change our function to

```
--Lingo
on squaredDistanceBetween x1, y1, x2, y2
  x = x1 - x2
  y = y1 - y2
  return (x * x + y * y)
end

//JavaScript
function squaredDistanceBetween( x1, y1, x2, y2 ){
  x = x1 - x2;
  y = y1 - y2;
  return (x * x + y * y);
}
```

and use the new test

```
--Lingo
spr = sprite(me.spriteNum)
if (squaredDistanceBetween(100, 100, spr.locH, spr.locV)<4)
  then
    --Do something
  end if

//JavaScript
spr = sprite(this.spriteNum);
if (squaredDistanceBetween(100, 100, spr.locH, spr.locV)<4){
  //Do something
}
```

Computationally this uses much less processing power, for no loss of accuracy. Always look out for times when you can lower the hit on the processor because your games will play more smoothly.

It doesn't seem very logical to me!

We've looked at how we can use the 'if ... then' structure to select which code to run. So far we have used a single condition. But we can combine conditions using either an 'And' operation or an 'Or' operation. Lingo uses the key word 'and' to represent And and 'or' to represent Or. JavaScript uses the symbol '&&' to represent And and '||' to represent Or. So how do we use these to combine conditions? The And option means that all the conditions must be true for the combined condition to evaluate to true. The Or option just requires a single condition to evaluate to true for the combined result to evaluate to true. Truth tables are often used to show the results of combinations. Table 7.2 shows the truth tables for both And and Or.

Table 7.2 *And (&&) and Or (||) truth tables*

And (&&)	True	False
true	true	false
false	false	false

Or ()	True	False
true	true	true
false	true	false

Let's look at how we can use these options to improve our conditional statements. Suppose we want to run a section of code if our character is inside a rectangle on screen. We will define the rectangle using the upper left corner and the lower right corner. To be within the region the character's 'locH' position must be greater than the value for left of the rectangle and also less than the value for right; in addition the character's 'locV' position must be greater than the top line of the rectangle but less than the bottom line. So how do we put this into code? Let's consider the rectangle (100, 200, 300, 400). Here the left side is at 100, the top is at 200, the right is at 300 and the bottom is at 400. If we have a sprite called `spr` then the code we would use would be

```
--Lingo
spr = sprite(me.spriteNum);
if (spr.locH>100 and spr.locV>200 and \
    spr.locH<300 and spr.locV<400) then
    --Do something
end if

//JavaScript
spr = sprite(this.spriteNum);
if (spr.locH>100 && spr.locV>200 &&
    spr.locH<300 && spr.locV<400){
    //Do something
}
```

This reads as 'if the locH position of sprite is greater than 100 and the locV position is greater than 200 and the locH position is less than 300 and the locV position is less than 400 then do something'.

But what if we want to 'do something' if the character is in one of two different rectangles. First let's put the test into a function.

```
--Lingo
on pointInRect x, y, left, top, right, bottom
    if (x>left and y>top and x<right and y<bottom) then
        return true;
    else
        return false;
    end if
end if
```

```
//JavaScript
function pointInRect( x, y, left, top, right, bottom){
  if (x>left && y>top && x<right && y<bottom){
    return true;
  }else{
    return false;
  }
}
```

This little function (see Chapter 6 for more information about creating and calling functions) returns true if the point (x,y) is inside the rectangle (left,top,right,bottom).

‘Examples/Chapter07/Lingo02.dir’ and ‘Examples/Chapter07/JavaScript02.dir’ show an example.

```
--Lingo
1 on enterFrame
2   put pointInRect (_mouse.mouseH, _mouse.mouseV, 100, 100,
   450, 300)
3 end
4
5 on pointInRect x, y, left, top, right, bottom
6   if (x>left and y>top and x<right and y<bottom) then
7     return true
8   else
9     return false
10  end if
11 end if

--JavaScript
1 function enterFrame(){
2   trace( pointInRect (_mouse.mouseH, _mouse.mouseV, 100, 100, ↵
   450, 300) );
3 }
4
5 function pointInRect( x, y, left, top, right, bottom){
6   if (x>left && y>top && x<right && y<bottom){
7     return true;
8   }else{
9     return false;
10  }
11 }
```

Listing 7.1

When running the example the message window shows 0 when the mouse is in the blue area and 1 when it is in the black area (see Figure 7.3).

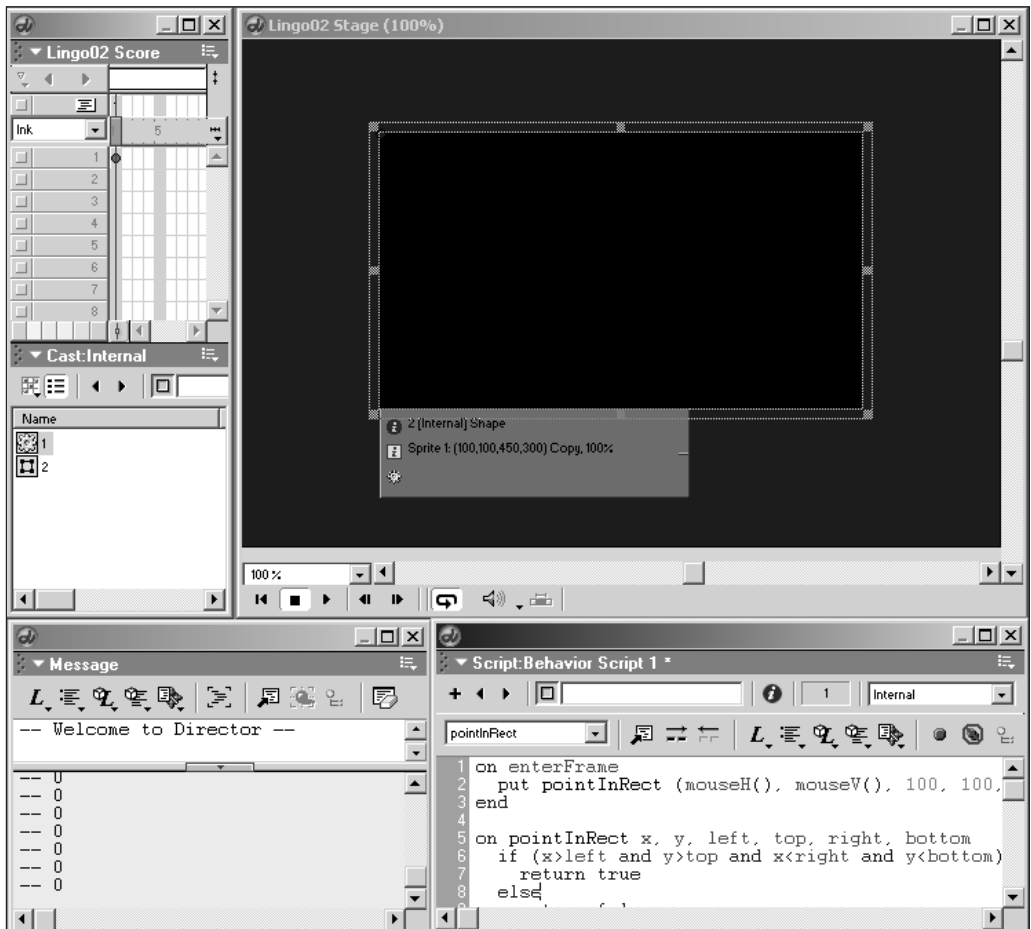


Figure 7.3 Developing the 'pointInRect' example

Developing the 'pointInRect' example

When you are learning about using any sort of programming language there is no substitute for hands-on experience. We are going to look at a practical example of the 'if ... then' structure in action and you are firmly encouraged to run your copy of Director and open 'Examples\Chapter07\Lingo03.dir' or 'Examples\Chapter07\JavaScript03.dir'. The purpose of the program is to detect when the mouse overlaps with the box in the middle of the screen. As well as detecting the overlap we set the light to be red or green (see Figure 7.4).

The example uses three sprites. The box with the rounded corners is Sprite 1, the caption is Sprite 2 and the circle is Sprite 3.

You can see from Figure 7.5 that the movie has just one frame. The script is a frame script on frame 1 of the script channel. You can also see that the movie contains an embedded font. The icon 'AaZz' indicates this. You can embed a font into a movie by selecting 'Insert/Media Element/Font...', which brings up a font selection dialog, as shown in Figure 7.6.

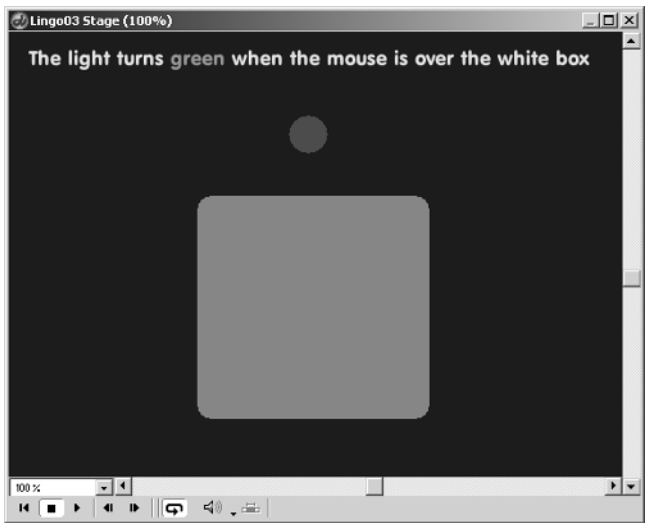


Figure 7.4 Enhancing the 'pointInRect' example

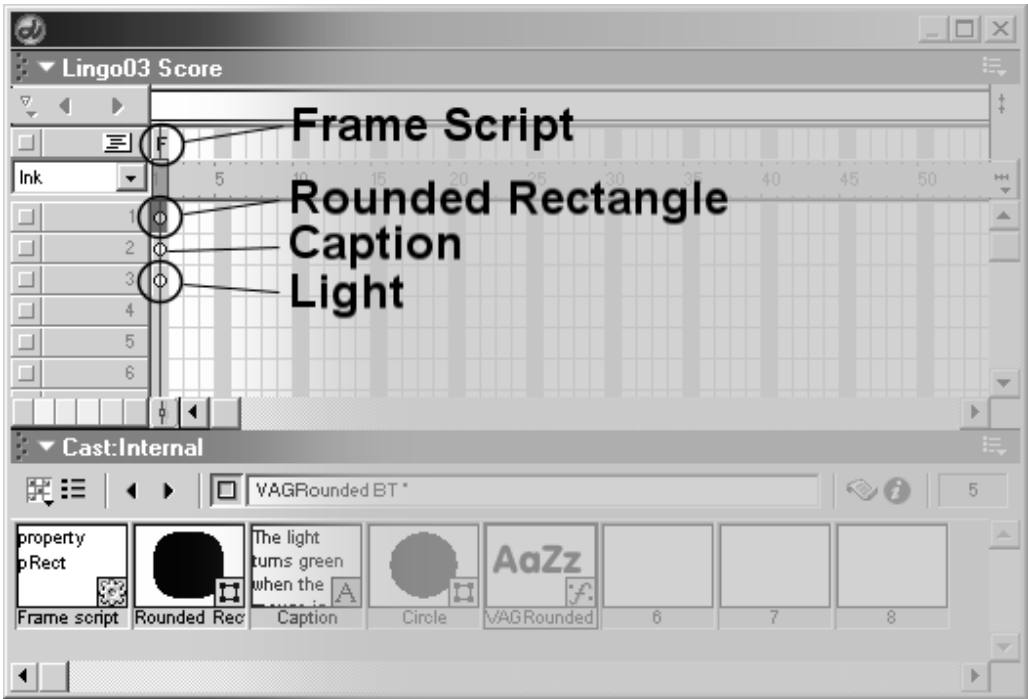


Figure 7.5 'Stage' and 'Cast' windows for 'Examples\Chapter07\Lingo03.dir'

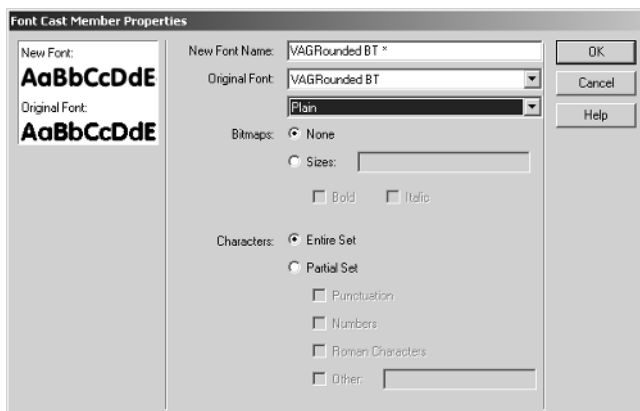


Figure 7.6 *Embedding a font*

The dialog allows you to select and view fonts. By default an embedded font takes the name of the original font and appends an asterisk (*). You can limit the amount of the font that is embedded to a subset using the 'Partial Set' option. Once you have embedded a font the text window will include this font name in the listing. To ensure you are working with the embedded font you must use the font with the alternative name. If a font is not embedded into a movie and the movie is then viewed on a computer that does not have the font installed then Director will substitute an alternative font instead. The designer would not be happy with the results, so make sure you get into the habit of embedding the fonts at an early stage. Listing 7.2 shows the code in the frame script. Note that the registration point for Sprite 1 must be positioned to the top left of the image for this code to work.

```
--Lingo
1 property pRect
2
3 on beginSprite
4   --Define the bounding rectangle
5   pRect = [#left:sprite(1).locH, #top:sprite(1).locV, \
6           #right:sprite(1).locH + sprite(1).width, \
7           #bottom:sprite(1).locV + sprite(1).height]
8 end
9
10 on enterFrame
11   if pointInRect (_mouse.mouseH, _mouse.mouseV, pRect) then
12     sprite(3).color = color(0, 255, 0)
13   else
14     sprite(3).color = color(255, 0, 0)
15   end if
16 end
```

```

17
18 on pointInRect x, y, r
19   if (x>r.left and y>r.top and x<r.right and y<r.bottom) then
20     return true
21 else
22   return false
23 end if
24 end if

//JavaScript
1 var pRect;
2
3 function beginSprite (){
4   pRect = new Array(sprite(1).locH, sprite(1).locV,
5                     sprite(1).locH + sprite(1).width,
6                     sprite(1).locV + sprite(1).height);
7 }
8
9 function enterFrame(){
10  if (pointInRect (_mouse.mouseH, _mouse.mouseV, pRect) ){
11    sprite(3).color = color(0, 255, 0);
12  }else{
13    sprite(3).color = color(255, 0, 0);
14  }
15 }
16
17 function pointInRect( x, y, r){
18   if (x>r[0] && y>r[1] && x<r[2] && y<r[3]){
19     return true;
20   }else{
21     return false;
22   }
23 }

```

Listing 7.2

First we define a property or 'var' in line 1. Recall that a property is a variable that remains valid throughout the life of a script. It is common practice and to be encouraged that properties of a script use the prefix p in the name, which ensures that whenever a variable of the form pXXXX appears it is immediately obvious when reading through the code that this is a property variable.

Note the symbol '\n' on the end of a line of script indicates that the line was too long when entering the code and the Alt + Enter was used to indicate a line that appears on the next line down but is treated as though the line is continuous. JavaScript code can be entered over multiple lines without using this facility.

A sprite can define a handler for when it is first loaded. This handler is called 'beginSprite'. We use this event to define the parameters of the rectangle, which are specified by the position and size of the sprite in channel 1. Note the use of '--' in line 4, meaning that anything on the same line following '--' is ignored; you can type anything in, the symbol is used to add a comment to the code. A comment helps describe a section of code. Comments are very useful when returning to some script. A sprite has many properties, its horizontal position is defined in the member variable 'locH' and the vertical position can be accessed using 'locV'. The width of the sprite can be read and so too can the height.

JavaScript

We define the rectangle as an array. JavaScript arrays are initiated using 'new'. Each member of the array can be accessed using an index number. So if we define an array using

```
myList = new Array(1, 3, "hello", new Array("this is a ↵
    sublist", true) );
```

then

```
myList[0] = 1;
myList[2] = "hello";
```

and

```
myList[3][1] = true.
```

'myList[3]' returns an array that can also be accessed using square brackets.

Lingo

We define the rectangle as a list. Lingo lists are initiated using square brackets. Each member of the list can be accessed using an index number. So if we define a list

```
myList = [1, 3, "hello", ["this is a sublist", true] ]
```

then

```
myList[1] = 1
myList[3] = "hello"
```

and

```
myList[4][2] = true.
```

'myList[4]' returns a list that can also be accessed using square brackets. When first creating a list you can give names to each of the members:

```
d = [#message:"this is a sublist", #update:true]
myList = [#first:1, #last:3, #greeting:"hello", #details:d]
```

Now we can use the names to access members:

```
myList.first = 1
myList.greeting = "hello"
myList.details.update = true
```

Names make reading the code much easier to understand but make no difference to performance. You are encouraged to make use of names in lists to make your code easier to maintain.

Code maintenance is an important issue. In large-scale teams, the structure of code and the documentation available to describe its inner workings are critical. By getting into good habits early on you will find that you naturally add comments to your code and have variable structures that help to describe what is happening to anyone who is trying to discover the workings of the script.

Lines 5–7 (Lingo) or 4–6 (JavaScript) of Listing 7.2 define the rectangle's left, top, right and bottom locations in pixels using the properties of Sprite 1. The data is stored in property 'pRect'. Once it is initialized in this way we can find the left top corner of the rectangle using 'pRect.left' (Lingo) or 'pRect[0]' (JavaScript) and 'pRect.top' (Lingo) 'or pRect[1]' (JavaScript).

Lines 18–24 (Lingo) or 17–23 (JavaScript) present a slightly changed version of the 'pointInRect' function that we examined earlier. The only change is the name of the variable passed as parameter 'r' to the function. This time the parameter is a list/array and can contain a multitude of information. The biggest change is to the 'enterFrame' handler; this time the 'if' condition is used to set the color of the sprite in channel 3. First we get the call to the function in line 11 (10, JavaScript). If this returns true, the following line, 12 (11, JavaScript), is executed. This line sets the 'color' property of Sprite 3 to 0 red, 255 green and 0 blue. Alternatively if the function returns false then the 'else' part is executed where the color is set to 255 red, 0 green and 0 blue.

Overlapping rectangles

See Figure 7.7.

Before we study the code let's try to understand the problem. The problem is to identify and classify the overlap of two rectangles. Figure 7.8 shows the possible locations of the two rectangles.

The smaller rectangle A can be positioned in one of 10 possible places, in relation to the larger B:

- 1 No overlap
- 2 Top left
- 3 Middle left
- 4 Bottom left
- 5 Top middle
- 6 Inside
- 7 Bottom middle
- 8 Top right
- 9 Middle right
- 10 Bottom right



Figure 7.7 Overlapping rectangles 'Example/Chapter07/Lingo04.dir'

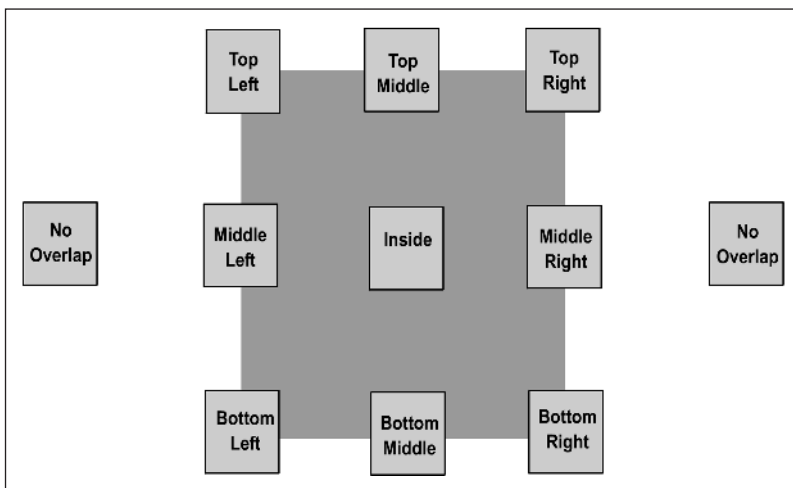


Figure 7.8 How two movie clips can overlap

If we were totally general and allowed the case where Sprite A can be bigger than B then there would be other cases, but for now we will only consider the case where both the width and height of Sprite (x) are less than Sprite (y).

Table 7.3 shows the conditions that will identify the placement of the smaller sprite in relation to the larger one; we use (left1, top1, right1, bottom1) to define the rectangle for A and (left2, top2, right2, bottom2) to define the rectangle for B.

Apart from the special case of 'no overlap' it takes four tests to determine the location: we need to test for the left, top, right and bottom of the first rectangle in relation to the second. The location determines which of these tests we need to do. If we first check the no overlap case and find

Table 7.3 *The conditions necessary to locate rectangle A in relation to rectangle B*

Location	left1	right1	top1	bottom1
No overlap	left1>right2	Or right1<left2	Or bottom1<top2	Or top1>bottom2
Top left	left1<left2	And right1>left2	And top1<top2	And bottom1>top2
Middle left	left1<left2	And right1>left2	And top1>top2	And bottom1<bottom2
Bottom left	left1<left2	And right1>left2	And top1<bottom2	And bottom1>bottom2
Top middle	left1>left2	And right1<right2	And top1<top2	And bottom1>top2
Inside	left1>left2	And right1<right2	And top1>top2	And bottom1<bottom2
Bottom middle	left1>left2	And right1<right2	And top1<bottom2	And bottom1>bottom2
Top right	left1<right1	And right1>right2	And top1<top2	And bottom1>top2
Middle right	left1<right1	And right1>right2	And top1>top2	And bottom1<bottom2
Bottom right	left1<right1	And right1>right2	And top1<bottom2	And bottom1>bottom2

that this does not apply then we need to identify which of the nine other cases is the current location. To do this we would test first for a column and then for a row in the column. For example, a successful test for (left1 < left2 and right1 > left2) would place rectangle A in the left column of Figure 7.8. Then we need to work out if we are in the top, middle or bottom row. This is achieved using the code snippet

```
--Lingo
...
else if (left1<left2 and right1>left2) then
  --Overlap to Sprite(y) left
  if (top1 < top2) then
    --Overlap at top left corner
    return corner
  else if (bottom1 > bottom2) then
    --Overlap at bottom left corner
    return corner
  else
    --Must be in middle on left
    return side
  end if
...
end if

//JavaScript
...
}else if (left1<left2 && right1>left2){
  //Overlap to Sprite(y) left
  if (top1 < top2){
    //Overlap at top left corner
    return corner;
  }else if (bottom1 > bottom2) {
```

```

        //Overlap at bottom left corner
        return corner;
    }else{
        //Must be in middle on left
        return side;
    }
}
...

```

The function returns a color value of red for no overlap, greenish yellow if at a corner, amber if at an edge and green if totally inside. This function is called by the `enterFrame` event of the purple-box sprite. Right click on the purple box and select 'Script...' to see the script.

```

1  property pMouse, pSprite
2
3  on beginSprite me
4      pos = [#x:_mouse.mouseH, #y:_mouse.mouseV]
5      pMouse = [#down:false, #pos:pos]
6      pSprite = sprite(me.spriteNum)
7  end
8
9  on mouseDown
10     pMouse.down = true
11     pMouse.pos.x = _mouse.mouseH - pSprite.locH
12     pMouse.pos.y = _mouse.mouseV - pSprite.locV
13 end
14
15 on mouseUp
16     pMouse.down = false
17 end
18
19 on enterFrame
20     if pMouse.down then
21         pSprite.locH = _mouse.mouseH - pMouse.pos.x
22         pSprite.locV = _mouse.mouseV - pMouse.pos.y
23         sprite(3).color = checkOverlap(sprite(1) )
24     end if
25 end
26
27 -----checkOverlap-----
28 -- Parameters
29 -- s - sprite to test
30 -- =====
31 -- Returns

```



```

32 -- red if no overlap
33 -- amber if intersecting
34 -- green if fully inside
35 -- =====
36 -- Description
37 -- Returns the overlap of two sprites
38 -- =====
39 on checkOverlap s
40   if pSprite.within(s) then
41     return color(0, 255, 0)
42   else if pSprite.intersects(s) then
43     return color(255, 200, 0)
44   else
45     return color(255, 0, 0)
46   end if
47 end

//JavaScript
1 var pMouse, pSprite;
2
3 function beginSprite( me ){
4   pMouse = new Array(false, new Array(_mouse.mouseH, _
      _mouse.mouseV));
5   pSprite = sprite(me.spriteNum);
6 }
7
8 function mouseDown(){
9   pMouse[0] = true;
10  pMouse[1][0] = _mouse.mouseH - pSprite.locH;
11  pMouse[1][1] = _mouse.mouseV - pSprite.locV;
12 }
13
14 function mouseUp(){
15   pMouse[0] = false;
16 }
17
18 function enterFrame(){
19   if (pMouse[0]){
20     pSprite.locH = _mouse.mouseH - pMouse[1][0];
21     pSprite.locV = _mouse.mouseV - pMouse[1][1];
22     sprite(3).color = checkOverlap(sprite(1));
23   }
24 }

```

```

25
26 //=====checkOverlap=====
27 // Parameters
28 // s - sprite to test
29 // =====
30 // Returns
31 // red if no overlap
32 // amber if intersecting
33 // green if fully inside
34 // =====
35 // Description
36 // Returns the overlap of two sprites
37 // =====
38 function checkOverlap( s ){
39   if (pSprite.within(s)){
40     return color(0, 255, 0);
41   }else if (pSprite.intersects(s)){
42     return color(255, 200, 0);
43   }else{
44     return color(255, 0, 0);
45   }
46 }

```

Listing 7.3

Here we have a more substantial block of code. It begins in the usual way by defining two properties and using a ‘beginSprite’ handler to initialize these two properties. Because we are using dragging we need to define a ‘mouseDown’ and a ‘mouseUp’ handler. These events occur when the mouse is clicked or released over the sprite. We set the variable ‘pMouse.down’ to true or false in relation to these events. Because we want the sprite to move based on where it is clicked we need to know how the click location relates to the sprite’s top left corner.

Figure 7.9 illustrates how we can anchor the dragging motion to the location of the first click. In the ‘mouseDown’ event we store the offset from the top left corner of the sprite to the mouse location using the code in lines 11 and 12 (10 and 11, JavaScript). On each frame we can use this offset to move from the mouse location to where the top left corner of the sprite should be positioned. The code in lines 21 and 22 (20 and 21, JavaScript) does just this. Note lines 27–38 (26–37, JavaScript) are a very full example of commented code. A comment in Lingo begins with the symbol ‘--’ and in JavaScript ‘//’. Anything that follows this symbol on the same line is regarded as a comment for the programmer’s use and is not included in the compiled movie. Maybe this is a little too complete for your project but it is the type of code demanded on very exacting large-scale team efforts. Line 40 (39, JavaScript) shows how we use a method of sprites, within, which returns true if the sprite is entirely inside the sprite passed as a parameter. In this instance it returns true if ‘pSprite’ is entirely inside ‘s’. Line 42 (41, JavaScript) shows yet another method of sprites, intersects. This method returns true if one sprite crosses over the other. If neither of these return true then there must be no overlap.

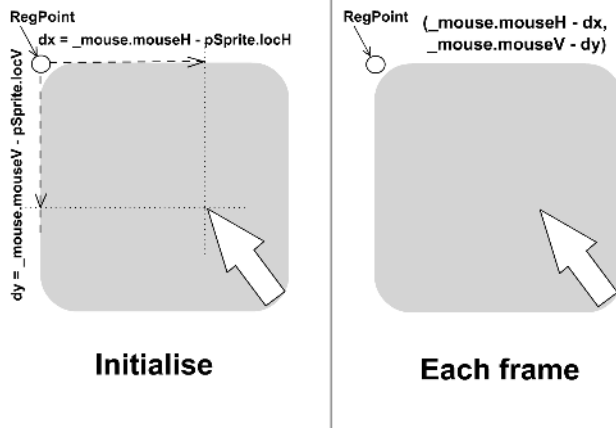


Figure 7.9 *Dragging sprites*

Writing your own overlap function

Maybe you need to be more precise with the overlap detection. You may want to know at which side or at which corner the overlap takes place. If this is the case then you will have to write your own overlap function and cannot rely on the methods of the sprite. On the CD 'Example/Chapter07/Lingo05.dir' shows how this can be done with Lingo and 'Example/Chapter07/JavaScript05.dir' is the JavaScript version. Listing 7.4 shows the revised code for the 'checkOverlap' function.

```
--Lingo
49 on checkOverlap s
50   left1 = pSprite.locH
51   left2 = s.locH
52   top1 = pSprite.locV
53   top2 = s.locV
54   right1 = left1 + pSprite.width
55   right2 = left2 + s.width
56   bottom1 = top1 + pSprite.height
57   bottom2 = top2 + s.height
58
59   corner = color(200, 255, 0)
60   leftright = color(255, 220, 0)
61   topbottom = color(150, 150, 255)
62
63   --These conditions indicate no overlap
64   if (left1>right2 or right1<left2 or top1>bottom2 \
65       or bottom1 < top2) then
66     pCaption.text = "No overlap"
```

```

67     return color(255, 0, 0)
68     else if (left1<left2 and right1>left2) then
69         --Overlap to s left
70         if (top1 < top2) then
71             --Overlap at top left corner
72             pCaption.text = "Top left corner"
73             return corner
74         else if (bottom1 > bottom2) then
75             --Overlap at bottom left corner
76             pCaption.text = "Bottom left corner"
77             return corner
78         else
79             --Must be in middle on left
80             pCaption.text = "Middle left"
81             return leftright
82         end if
83     else if (left1<right2 and right1>right2) then
84         --Overlap to sprite s right
85         if (top1 < top2) then
86             --Overlap at top right corner
87             pCaption.text = "Top right corner"
88             return corner
89         else if (bottom1 > bottom2) then
90             --Overlap at bottom right corner
91             pCaption.text = "Bottom right corner"
92             return corner
93         else
94             --Must be in middle on right
95             pCaption.text = "Middle right"
96             return leftright
97         end if
98     else
99         --Must be in middle column
100         if (top1<top2) then
101             --Middle top of sprite s overlap
102             pCaption.text = "Middle top"
103             return topbottom
104         else if (bottom1>bottom2) then
105             --Middle bottom of sprite s overlap
106             pCaption.text = "Middle bottom"
107             return topbottom
108         else
109             --Must be completely inside sprite s

```

```
110      pCaption.text = "Inside"
111      return color(0, 255, 0)
112    end if
113  end if
114 end
```

Listing 7.4

The line numbers start at 49 simply because the numbers correspond to Director line numbers and this listing excludes the start of the code. The function begins by initializing some useful local variables; `left1`, `left2` and so on. These variables are then used in multiple conditions to determine the location of 'pSprite' in relation to 's'. Read through the listing to work out how the various tests work.

Summary

The 'if ... then' structure is one of the most useful structures you will find in any programming language. It allows you as developer to select the code that will run based on current conditions. In this chapter we looked at using the if ... then structure in both simple and complex ways. The if ... then structure takes a parameter that must evaluate to either 'true' or 'false'. We looked at using simple conditions and also multiple conditions using the combining logic of 'And' and 'Or'. In the next chapter we look at how we can execute repetitive tasks easily.

8 Using loops

If there is one thing that computers are good at, although if yours has just crashed on you then you might be forgiven for thinking they are good at precious little, it is doing boring repetitive jobs. In this chapter we will look at how you can tell your program to repeat and repeat and repeat. No programming language would be complete without the ability to repeat sections of code several times. Games regularly need to run a section of code several times. In this chapter we will look at creating sections of code that repeat and show you how to control the program flow in these situations. We will learn how to jump out of the code if a certain condition is met. We will consider using repeats that run at least once and repeats that may never run at all. One of the most important aspects of using loops is the initializing of the data used in a loop. We will look at the potential pitfalls from initialization errors.

Creating sections of code that repeat

Director has several options for repeating; the first option is used when you already know how many times you want to repeat a section of code. In this circumstance you use ‘repeat with...’ using Lingo or a ‘for’ loop using JavaScript.

```
--Lingo
1  on startMovie
2      total = 0
3      repeat with i= 1 to 10
4          total = total + i
5      end repeat
6      put "The sum of the numbers from 1 to 10 is " & total
7  end

1  function startMovie(){
2      var total = 0;
3      for(var i=1; i<=10; i++){
4          total += i;
5      }
6      trace( "The sum of the numbers from 1 to 10 is " + total);
7  }
```

Listing 8.1

Using this construction we use a simple variable, in this case ‘i’. This variable is initialized to the value following the equals sign and for each run through the loop, 1 is added to its value. ‘Examples/Chapter08/Lingo01.dir’ or ‘Examples/Chapter08/JavaScript01.dir’ shows this code in action. Notice the code in line 4 of the JavaScript version; this means add and assign. In other words

Table 8.1 *Each loop of the program
‘Examples/Chapter08/XXX01.dir’*

Loop number	i	Total
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
6	6	21
7	7	28
8	8	36
9	9	45
10	10	55

add i to the variable ‘total’ and assign the value to total. It is a shorthand version of the code in line 4 of the Lingo version. Most operators have an assign version as well as their more familiar form. You are likely to see this form regularly when reading through JavaScript code listings in magazines or on the web. The ‘for’ statement takes three parameters, the first is the initialization case, the second is a test that is applied for each loop and the third and final parameter is what is done at the end of each loop. Usually the final parameter adds 1 to the main loop variable. As you know the symbol ‘++’ means add 1 to the variable name that precedes it.

The variable used can be decremented rather than incremented if required using the construction in Listing 8.2.

```
--Lingo
1  on startMovie
2    total = 0
3    repeat with i= 10 down to 1
4      total = total + i
5    end repeat
6    put "The sum of the numbers from 1 to 10 is " & total
7  end

//JavaScript
1  function startMovie(){
2    var total = 0;
3    for (i=10; i>0; i--){
4      total += i;
5    }
6    trace( "The sum of the numbers from 1 to 10 is " + total);
7  }
```

Listing 8.2

Table 8.2 *Each loop of the program
'Examples/Chapter08/Lingo02.dir'*

Loop number	i	Total
1	10	10
2	9	19
3	8	27
4	7	34
5	6	40
6	5	45
7	4	49
8	3	52
9	2	54
10	1	55

Using this technique the variable 'i' is initially set to the value 10 and then reduced by 1 for each loop. Table 8.2 shows the result.

Jumping out of the code if a condition is met

Sometimes you may want to loop until some condition is met.

```
--Lingo
1  on startMovie
2      total = 0
3      i = 1
4      repeat while total<=1000
5          total = total + i
6          i = i + 1
7      end repeat
8      put "The sum of the numbers from 1 to " & i & " is " & total
9  end
```

```
//JavaScript
1  function startMovie(){
2      var total = 0;
3      var i = 1;
4      while (total<=1000){
5          total += i;
6          i++;
7      }
8      trace( "The sum of the numbers from 1 to " + i + " is " + ↵
          total);
9  }
```

Listing 8.3

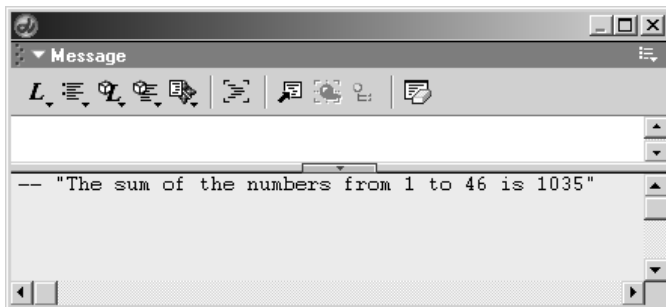


Figure 8.1 Output from the movie 'Examples/Chapter08/XXX03.dir'

In this example we use the 'while...' construct, which is followed by a condition that evaluates to true or false. Because we start with 'total' and 'i' set to zero it adds the value of i to the total until the total is greater than 1000.

Skipping part of the repeating block of code

Another alternative that may suit your problem is

--Lingo

```
1  on startMovie
2    total = 0
3    i = 1
4    repeat while true
5      if (total + i)>1000 then exit repeat
6      total = total + i
7      i = i + 1
8    end repeat
9    put "The sum of the numbers from 1 to " & i & " is " & total
10 end
```

//JavaScript

```
1  function startMovie(){
2    var total = 0, i = 1;
3
4    while (true){
5      if ((total + i)>1000) break;
6      total += i;
7      i++;
8    }
9    trace( "The sum of the numbers from 1 to " + i + " is " +
           total);
10 }
```

Listing 8.4

This example, 'Examples/Chapter08/Lingo04.dir' or 'Examples/Chapter08/JavaScript04.dir', has the condition set to true always. However, within the repeat loop you get an 'if' statement. Line 5 tests whether adding 'i' to the value of 'total' would result in a number greater than 1000; if so then the 'exit repeat' code or 'break' is executed, the result of which is to jump out of the loop to line 9.

Ensuring the exit condition

One of the most common problems with repeat loops is getting stuck in an infinite loop. Take a look at Listing 8.5. This is nearly identical to Listing 8.4, the only difference being that line 5 is missing.

```
--Lingo
1  on startMovie
2    total = 0
3    i = 1
4    repeat while true
5      total = total + i
6      i = i + 1
7    end repeat
8    put "The sum of the numbers from 1 to " & i & " is " & total
9  end

//JavaScript
1  function startMovie(){
2    var total = 0, i = 1;
3
4    while (true){
5      total += i;
6      i++;
7    }
8    trace( "The sum of the numbers from 1 to " + i + " is " +
          total);
9  }
```

Listing 8.5

Now no condition will cause the loop to terminate. Similar problems can result from not incrementing a variable (see Listing 8.6). In this instance we are waiting for 'i' to be greater than 10, but nothing is changing i so it will stick on 1 as many times through the loop as it can.

```
--Lingo
1  on startMovie
2    i = 1
3    repeat while i<10
4      put i
5    end repeat
6  end
```

```
//JavaScript
1  function startMovie(){
2      var i = 1;
3      while (i<10){
4          trace( i );
5      }
6  }
```

Listing 8.6

Another common problem is using the equal case for a test; take a look at Listing 8.7. Here we are looking for $i = 10$; the code will repeat if i is not equal '<>' (or in JavaScript ' \neq ') to 10. But the variable is being incremented by itself, so as you can see from the output given in Table 8.3, i is never equal to 10 in the loops. Making the condition $i \geq 10$ would rectify the problem. The loop would exit with i equal to 16 on loop number 5.

```
--Lingo
1  on startMovie
2      i = 1
3      repeat while i<>10
4          i = i + i
5      end repeat
6  end

//JavaScript
1  function startMovie(){
2      var i = 1;
3      while (i!=10){
4          i += i;
5      }
6  }
```

Listing 8.7

Table 8.3 Each loop of Listing 8.7

Loop number	i	i + i
1	1	2
2	2	4
3	4	8
4	8	16
5	16	32
6	32	64
7	64	128
8	128	256
9	256	512
10	512	1024

Pitfalls caused by initialization errors

When using the loop construction you will find that the most common problem is due to inappropriate initialization. Take the following example:

```
--Lingo
1  repeat while i<>20
2    --Do something
3    ...
4    i = i + 2
5  end repeat
```

```
//JavaScript
1  while (i<>20){
2    //Do something
3    ...
4    i += 2;
5  }
```

Listing 8.8

Here we enter the loop with no initialization, blindly assuming that ‘i’ will default to zero. Never assume anything about initialization. You will find that almost all hard-to-locate errors will be the result of poor initialization. How do I know? Because I have done most of them! Don’t be stupid like me; make sure that all variables that are used inside a code loop have a known and predictable value at the start and throughout the code loop. The example above also betrays another common problem with loops using any kind of equality condition; in the above we allow the loop to continue until i is equal to 20. But, what if i will never equal 20? Maybe it starts equal to 25. Therefore, as i increases for every iteration of the loop, it will never equal 20. You may find an instance where i starts the loop as an odd number. When 2 is added to an odd number the number stays odd, so again it will never equal 20, which, as you know, is an even number. Always use greater than or less than as the exit condition; you will eliminate at least half the problems you may encounter with loops by adopting this useful advice.

Summary

In this chapter we looked at the third vital component of all programming languages, loops. With this third important component you are nearly ready to get on and start writing your own code. You might think, great, I’m ready, or might think how do I move from a game idea to a finished game? Planning and structure are the answer. Forward planning and good structure will make all the difference to the development of your next masterpiece. So before setting you loose I recommend spending some time reviewing the tips in the next chapter. There we look at how you can structure your programs to make your games easier to write, debug and maintain and you will see how you can use bits from one game in another.

9 Keep it modular

First, congratulations on your tenacity. If you have worked through the book and got as far as this chapter then you are well on your way to becoming a fully-fledged Director developer. In this chapter we are going to introduce the skills you will need to really develop your own games rather than just copying someone else's code. This is an entire chapter with advice on how to structure your code from an idea to a game. Throughout we will consider the example of writing a 'Tetris' game. Unlike most chapters in the book there is not a complete working example, instead you are encouraged with hints to develop your own version of the game.

Like most programming environments, Director can provide you with the programming tools to make your code well structured. Unfortunately, this flexibility can also be used to create very poorly structured code. Game development can be made very hard or very easy. Careful planning and good structure makes development so much easier. In this chapter we look at how to take an idea for a game, turn it into a well-planned project and then develop the code. We will look at using custom functions to make sure that you only have to change your code in one place to have a global effect. We look at using functions to change the value of certain variables. We look at the difference between local and global variables. But most importantly we will look at how to embed complexity into your behaviors so that they can look after themselves and better still you will have code that you can reuse.

From an idea to a plan

The starting point for all games is an idea. Your idea is the basis for all subsequent development. Because we are concerned with games let us consider a game that everyone will be familiar with – 'Tetris'. In Tetris we have a board layout that is based on square cells. Each cell can either have a tile in it or not. At its simplest that is the game. But the game play is another matter. The game revolves around a block of four tiles in a line, a square block, a T-shape or an L-shape dropping at a consistent rate from the top. When any one of the four tiles that make up the set of tiles hits the base of the play area the movement of that set of tiles stops and the set of tiles remains on the screen. At this point we need to create a new block of tiles and start this dropping. This new set of tiles and all subsequent ones stop when either the base is reached by any one of the tiles or any one of the tiles hits a parked tile.

Over the course of the game the speed at which the tiles drop increases. Parked tiles are removed if, as the current dropping set of tiles lands, a complete row of tiles is created. By removing tiles the score increases and if the user simultaneously creates more than one row then they get a higher score than a pro-rata multiple of the value of a single row. The maximum number of rows that can be removed in a single removal is four, which results in the maximum additional score.

Does that sound like a full description of 'Tetris'? If it does then it only indicates how easy it is to overlook a very important feature. As described above the user does absolutely nothing! Such a game is unlikely to become a world beater. The input from the user to control the horizontal position of the descending tile set has been overlooked in the summary. When you are writing

your plan you need to work extremely hard to ensure that you have considered all the options. But the single most important consideration when writing the specification for a game is to decide what the user actually does when playing your game. If all they do is to admire your lovely graphics then go back to the drawing board.

From a plan to a structure

Now we have a plan, but how do you intend to turn this into a working game? A very popular concept in computer science is to use the top-down approach. You start by presenting an overview of the game. Then you break the game down into manageable chunks that can be developed in isolation. This methodology is a very effective technique when computer programming; that is, creating chunks of an application that can be developed in isolation and debugged separately.

Returning to our ‘Tetris’ example, let’s try to break down the overview into small chunks of functionality.

- *Initialization* – set up any variables that the game is likely to need.
- *New game* – when all the game variables are initialized and the game board is cleared.
- *User input* – we are going to need a keyboard reader that moves the playing piece if such a movement is possible.
- *Next piece generator* – we will need to be able to create the next piece that is going to be used in the game, which randomly determines the next piece to fall. The game is over when a new tile cannot be added, so if this function returns false then this indicates the end of the game.
- *Legal move confirmation* – we will need to be able to confirm whether the descending tile can move to a certain game board position.
- *Complete row checker* – we will need to be able to confirm when a completed row has occurred.
- *Complete row removal* – this is where the game board will be adjusted to remove a completed row.
- *Update game board* – the game board is controlled using a multi-dimensional array.
- *Game over* – this could be a complex animation or a simple text box could be displayed.
- *Timer update* – if the score is going to use elapsed time as well as removed rows then we will need to show a timer.
- *Score update* – the score should be continuously updated based on the rows removed and the elapsed time.

We are also going to need some variables to track the game’s progress:

- *pScore* – the current player’s score.
- *pTiles* – a list of lists giving information about the current piece and how to display it.
 - *#tiles* – this is a $4 \times 4 \times 4$ multi-dimensional list, which describes the current tile; the possible values for this are illustrated in Table 9.1. Each tile set can be rotated, so, for example, the strip of four vertical tiles when rotated becomes a strip of horizontal tiles. There are four different tile sets, strip (1), block (2), L-shaped (3) and T-shaped (4).
 - *#typeID* – which of the tiles lists we are currently using.
 - *#rotID* – which tile rotation in the tile-type lists we are using.
 - *#x* – the x-position within the board where the descending tile is located. This is where the left column of the descending tile maps.

- #y – the y-position within the board where the descending tile is located. This is where the top row of the descending tile maps.

If an L-shaped tile set is being used, where there are three horizontal tiles and the extra tile is below the rightmost tile, the whole set being positioned at 5 horizontally and 6 vertically, we would have

```
pTile.typeID = 3 -- L-shaped set
pTile.rotID  = 2 -- Second in the rotation set
pTile.x      = 5
pTile.y      = 6
```

A list of lists creates a multi-dimensional list and an array of arrays creates a multi-dimensional array. For example, the following code

Lingo

```
fruits = ["oranges", "lemons", "apples"]
veg = ["potatoes", "cabbages", "carrots"]
food = [fruits, veg]
repeat with i=1 to food.count
    repeat with j=1 to food[i].count
        put "food[" & i & "]"[" & j & "]=" & food[i][j]
    end repeat
end repeat
```

JavaScript

```
fruits = new Array("oranges", "lemons", "apples");
veg = new Array("potatoes", "cabbages", "carrots");
food = new Array(fruits, veg);

for (var i=0; i<food.length; i++){
    for (var j=0; j<food[i].length; j++){
        trace( "food[" + i + "]"[" + j + "]=" + food[i][j]);
    }
}
```

generates the following output

```
-- "food[1][1]=oranges"
-- "food[1][2]=lemons"
-- "food[1][3]=apples"
-- "food[2][1]=potatoes"
-- "food[2][2]=cabbages"
-- "food[2][3]=carrots"
```

Table 9.1 Values stored in multi-dimensional arrays to store the current tile

1	0	0	0	1	1	1	1	1	1	0	0				
1	0	0	0	0	0	0	0	1	1	0	0				
1	0	0	0	0	0	0	0	0	0	0	0				
1	0	0	0	0	0	0	0	0	0	0	0				
1	1	0	0	1	1	1	0	0	1	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	1	0	0	1	1	1	0
1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	1	0	0	1	0	0	0
0	1	0	0	1	1	0	0	1	1	1	0	1	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

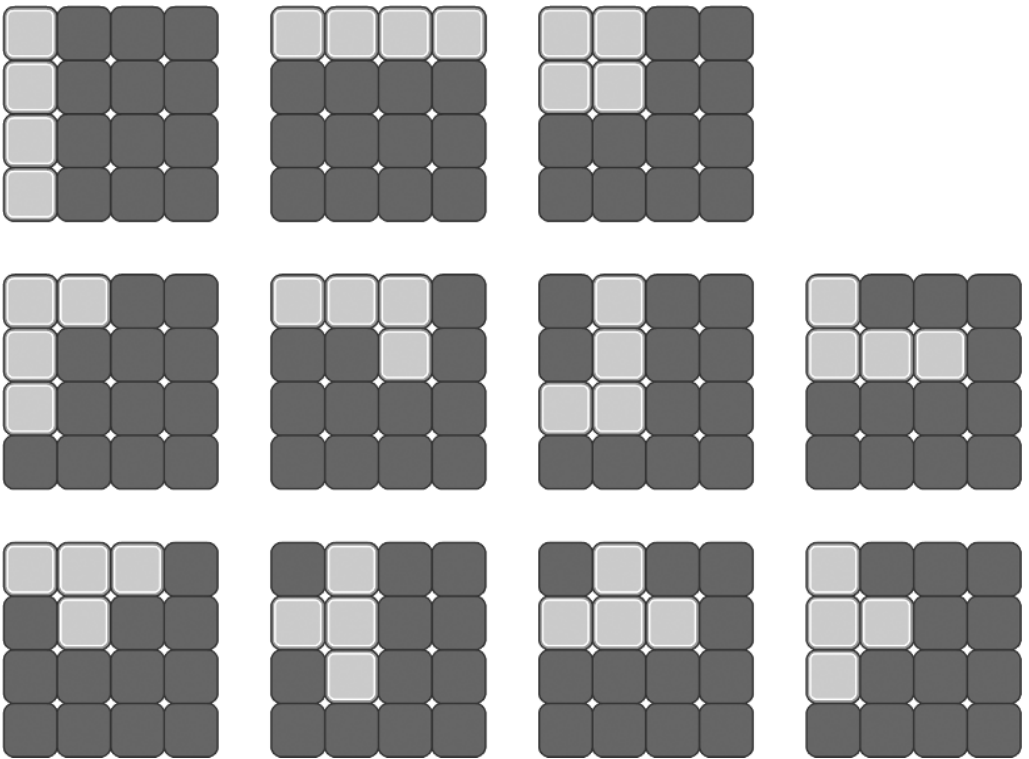


Figure 9.1 How the values in Table 9.1 translate into images in the game

- *pBoard* – this is a multi-dimensional list that stores the current board position. A value of ‘1’ in a cell of the list indicates a tile is present and a value of ‘0’ means it is empty.
- *pStartTime* – useful for generating the time elapsed since the start of the game.

Multi-dimensional lists are very useful when a program lends itself to a grid-like structure. We are basing the code for the ‘Tetris’ example on a grid-like structure; we will use 21 rows and

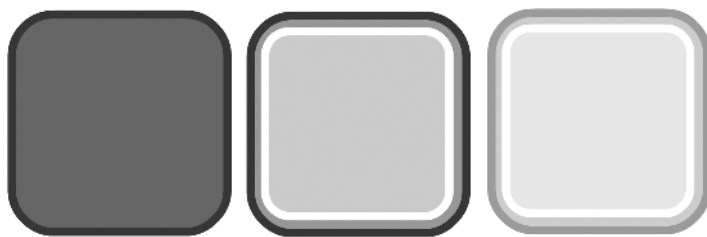


Figure 9.3 Possible displayed images for each cell in the game board grid

```

4   initBoard
5   newGame
6   end
7
8   on exitFrame me
9     checkKeys
10    updateBoard
11    checkRows
12    drawBoard
13  end
14
15  on initBoard
16    pBoard = []
17    repeat with x=1 to 14
18      pBoard.add([])
19      repeat with y = 1 to 21
20        pBoard[x].add(0)
21      end repeat
22    end repeat
23  end
24
25  on newGame
26    pScore = 0
27  end
28
29  on checkKeys
30  end
31
32  on nextPiece
33  end
34
35  on checkMove
36  end

```

```

37
38 on checkRows
39 end
40
41 on removeRows
42 end
43
44 on updateScore
45 end
46
47 on updateTimer
48 end
49
50 on drawBoard
51   img = _movie.stage.image
52   srcRect = rect(0, 0, 18, 18)
53   repeat with x=1 to 14
54     left = (x-1) * 18
55     repeat with y=1 to 21
56       top = (y-1) * 18
57       if pBoard[x][y]=1 then
58         tile = _movie.member("tiles0002").image
59       else
60         tile = _movie.member("tiles0001").image
61       end if
62       img.copyPixels(tile, \
63         rect(left, top, left + 18, top + 18), \
64         srcRect)
65     end repeat
66   end repeat
67 end
68 on updateBoard
69   repeat with x=1 to 14
70     repeat with y = 1 to 21
71       pBoard[x][y] = random(2)-1
72     end repeat
73   end repeat
74 end
75
76 on gameOver
77 end

```

Listing 9.1

The purpose of the function ‘initBoard’ is to populate the list of lists ‘pBoard’ with all zero entries. A list can be initialized using the code

```
myList = []
```

Then it can be appended using

```
add
```

In this function between lines 15 and 23 we use a nested repeat loop. The first repeat loop increments through the ‘x’ values, creating a new sublist with each iteration. The nested repeat loop adds 21 zeros to the list.

Most of the code in the example just gives place holders that will be used to add the functionality. One function that is already written for you is the ‘drawBoard’. This function will be called every frame loop and needs some explanation.

Line 51 sets a reference variable ‘img’ to point at the current stage image. This allows us to paint the stage without a single sprite on the score. Then we create a useful rectangle in line 52. The tiles that make up the board are all 18-pixel squares. This rectangle is set to an 18×18 square positioned at (0, 0). Then we enter a repeat loop where a variable called ‘left’ is initialized to the current value of the ‘x’ variable minus 1, multiplied by 18. This will create a series of values starting at zero and going up in intervals of 18. Line 53 is a nested loop for the ‘y’ values. Here a variable called ‘top’ is set to the value of y minus 1, multiplied by 18; again a series of values starting at 0 and increasing in increments of 18. Lines 57–61 select the image to display for the current state of the board at this square. Finally, lines 62 and 63 are an overlength single line (recall that to enter a long line use Alt + Enter); this uses the ‘image’ method ‘copyPixels’. This method takes three or four parameters. The first parameter is the image we are going to copy pixels from. The second parameter is the destination rectangle; here we take the current left and top position. The third parameter in this example is the source rectangle that we initialized as the variable ‘srcRect’. The fourth parameter, if used, can adjust the way the copy operates, semi-transparency is one option.

To show the effect the function ‘updateBoard’ has two repeat loops that simply randomly set the board tiles. If you run the movie then you will see the board flashing rapidly.

By making your program modular using functions, you place complex functionality in a black box, which you can test and then use elsewhere in your program. In the above example the function ‘updateGameBoard’ allows you to update the board at any time simply by calling the function.

To call a function, simply place the name of the function in your script, along with any parameters the function needs. To call ‘updateGameBoard()’ you would use

```
...
updateGameBoard()
...
```

From a project to a game

Already the project is developing into a game. Remember that a function must do exactly what it is intended to do regardless of the current state of the game or passed parameters. If passed parameters are unsuitable then pass this information on either by using a 'put' or by using an on-screen text box, which will ultimately be hidden from the player but will make creating the game easier for you as the developer. When you develop new functions you can place them all inside the same script. When developing Director games we tend to place most functions in a single behavior.

You will never succeed in creating complex games by writing the whole game and then hoping that it works. It simply doesn't work like that. Modularize the game. Write each individual component and then test the components using test data. When you set out to do the testing, you should try to break the function using extreme data parameters. In game play the extremes are likely to occur at some stage. Try to keep all the functionality for an aspect of your game in one place. Director is very flexible; you could have a condition inside a behavior that sets the variables in another script. When you return to a project that is set up in this way it is very difficult to work out what is happening. An example may illustrate the problem more effectively. Imagine a kids' maths game. There is a cartoon character holding up a board on which you need to enter a value. If you get it wrong then the board is held up again; if you get it wrong a second time then the cartoon character shows you the correct answer. We could put all the functionality into the cartoon character behavior script. The script could initialize a 'pWrongCount' property at the start of a problem then increment the value if the child gets the sum wrong. That method would be fine. Problems are likely to occur if a frame script controls some aspects of the behavior of the sprite but updating of the 'pWrongCount' property is done by the sprite behavior. Although this method could certainly be made to work it is likely to confuse the developer when they return to the game to tweak it, some weeks later. Either control everything at the frame-script level or control everything in the behavior. If everything is controlled inside the behavior then why not add a 'do nothing' script to the first frame of the movie.

```
--Functionality is control inside the Rhino sprite (channel 1)
--The property 'pWrongCount' is update everytime the child
--enters an incorrect value. This is handled using the ␣
    'enterData' loop
--When the child presses the return key the 'checkKeys' ␣
    function calls
--'validateData'. A correct answer sets 'correct' to true
--An incorrect answer and 'correct' is set to false
--and 'pWrongCount' is incremented. If 'pWrongCount' is ␣
    equal to one
--then the script send playback to the 'showAnswer' marker
--where the correct answer is displayed for the child.
--Then the game moves on to the next problem by calling ␣
    'nextProblem'
--and jumping to the main 'gameLoop' marker
```

Believe me, if you do that you will feel a warm glow when you return to your own code and other developers who read your code will think just one thing – respect!

One final tip – don't change global values directly. If you find that you do need to change the value of a global variable in several places then set the variable's value using a function, such as

```
on setMyGlobal (newValue, calledFrom)
    put "setMyGlobal to " & newValue & " called from " &
        & calledFrom
    myVariable = newValue
end
```

This will make tracking down any errors much easier. The 'put' will show you who attempted to change the value of the variable. Often you can find that as you develop your programs there may be errant code, which is left over from an earlier incarnation of the program and that this errant code is setting the values in a sprite behavior inappropriately. These types of errors are much easier to find if you follow the simple rule: 'Don't change the value of a variable directly, outside the current scope. Always use a function call that says who was trying to change the value.'

Creating a new object using Lingo

Object-orientated programming involves combining the data and the operations on the data within a single object. Director allows you to create objects by creating new instances of a script. In this code snippet we create a script called 'point' (see 'Examples/Chapter09/Lingo02.dir' for the code). Listing 9.2 shows the Lingo. It uses a handler for 'new'. This handler can have any number of arguments but the first is a reference to itself. It doesn't have to be the name 'me'; in many programming languages the word 'this' is used. If you choose you can use the word this as long as it is used as the first argument. The new handler must return the value of this first argument. You can have any number of functions in the script; these will become available to an instance of the script.

```
1  property x, y
2
3  on new me, xx, yy
4      x = xx
5      y = yy
6      return me
7  end
8
9  on add me, p
10     x = x + p.x
11     y = y + p.y
12 end
13
14 on dump
15     put "(" & x & ", " & y & ")"
16 end
```

Listing 9.2

To create an instance of the script use the code in Listing 9.3.

```

1  on startMovie
2    pt1 = new(script "point", 10, 20)
3    pt2 = new(script "point", 100, 50)
4    pt1.dump()
5    pt2.dump()
6    pt1.add(pt2)
7    pt1.dump()
8  end

```

Listing 9.3

Lines 2 and 3 create two instances of the script assigned to the variables 'pt1' and 'pt2'. The parameters we pass will be used in the new handler. For example, in line 2 the first parameter will be used as 'xx' in the handler and the second as 'yy'. Line 4 then uses the function in the script 'point', 'dump', to place information about the instance to the 'Message' window. Line 6 then uses the function 'add' to add the two points together. The results are displayed in Figure 9.4.

Object-orientated programming is used extensively in major software projects and is well worth trying in your own code. Although it doesn't allow you to do anything you can't already do with procedural techniques, by combining data and functionality it is often easier to create robust code.

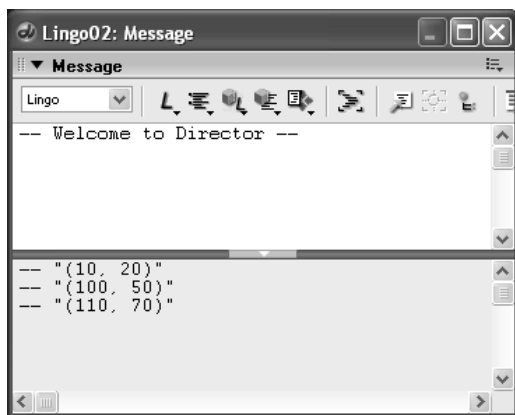


Figure 9.4 Output from the program
'Examples/Chapter09/Lingo02.dir'

Creating a new object using JavaScript

JavaScript has many more object-orientated programming features than Lingo. The simplest technique is to create an object that has accessible properties. This is easily achieved using the syntax in Listing 9.4.

```

function startMovie(){
  var pt1 = { x:1, y:2 };
  var pt2 = { x:5, y:6 };
  trace("Point (" + (pt1.x + pt2.x) + ", " + (pt1.y + pt2.y) + ")");
}

```

Listing 9.4

Any properties defined between the curly braces become accessible using dot syntax.

The next stage is to define a class using a constructor function. A constructor function takes the name of the class and then initializes the member variables of this class. If this sounds complicated then take a look at the code in lines 8–11 of Listing 9.5. Line 8 defines a ‘Point’ function, then the following two lines take the two parameters and assign them to the instance of this Point object using the keyword ‘this’. When the code in line 2 is called, Director creates a new object by calling the Point function and sets the values of ‘x’ and ‘y’ in this instance of a Point to the values passed to the following function.

```

1  function startMovie(){
2      var pt1 = new Point(1, 2);
3      var pt2 = new Point(5, 6);
4      trace("Point (" + (pt1.x + pt2.x) + ", " + (pt1.y + pt2.y) +
          ")");
5  }
6
7  // Point constructor function
8  function Point(x, y) {
9      this.x = x;
10     this.y = y;
11 }
```

Listing 9.5

Now we can create a Point object by using the ‘new’ key word. This calls the function and runs whatever code is inside the function call.

An important feature of object-orientated programming is to include the operations that you may apply to the data of an object within the object itself. So as well as being able to access the data components using dot syntax you can also call methods using the same syntactic style.

```

1  function startMovie(){
2      var pt1 = new Point(1, 2);
3      var pt2 = new Point(5, 6);
4      var pt = new Point();
5      pt.add(pt1, pt2);
6      pt.dump();
7  }
8
9  // Point constructor function
10 function Point(x, y) {
11     this.x = x;
12     this.y = y;
13 }
14
15 // add two Point objects
```



```

16 Point.prototype.add = function(a, b) {
17     this.x = a.x + b.x;
18     this.y = a.y + b.y;
19 }
20
21 // Trace the content of a Point object
22 Point.prototype.dump = function(){
23     trace("Point (" + this.x + ", " + this.y + ")");
24 }

```

Listing 9.6

Notice that in addition to the constructor function defined between lines 10 and 13 of Listing 9.6 we now have two further functions that use a different syntax. Line 16 introduces the prototype key word. A ‘prototype’ is followed by a method name. This is set to a function with an arbitrary number of parameters, in this instance two. The two parameters passed in line 16 are both ‘Point’ objects and as such have member variables called ‘x’ and ‘y’. We take the values of these two variables, add them together and assign them to the current instance of the Point. So in line 2 we create a Point object by calling the function defined at line 10. In line 3 we create second Point object using the same function call but this time with different parameters. In line 4 we create a Point with undefined data, but in line 5 we take the two Points we defined in lines 2 and 3 and call the function at line 16, which assigns the sum of these two Points to the current Point, pt. Finally we call the second function defined in line 22, which outputs the content of the Point to the message window using the ‘trace’ method.

Using classes in this way makes your code more robust, easier to read and easier to maintain and you are strongly urged to adopt these techniques in your own code.

Object-orientated programming is used extensively in major software projects and is well worth trying in your own code. Although it doesn’t allow you to do anything you can’t already do with procedural techniques, by combining data and functionality it is often easier to create robust code.

Summary

In this chapter we looked at how to modularize your code so that you can write small chunks of functionality that can be used from anywhere in your game. We looked at how you can break down the way a game will be coded by splitting the game into small segments and identifying key variables. Functions are such a useful idea that they should form the basis of much of the more complex behavior of your games. ‘enterFrame’ or ‘exitFrame’ handlers can be used for reading user input or navigation, but they should not be used for more complex testing such as a collision detection or data formatting. This should always be put inside a function so that it can be called from different places in your game. Remember, write it once and use it lots of times. You will often find developers who place almost the same code in many places in a program; if the code needs tweaking then they must change all the instances of it, a technique prone to error. When designing your games try to keep all the functionality for a particular aspect of your game in one place. Many of the techniques in this chapter are designed to make your programs easier to write, easier to extend and easier to debug. This last aspect leads on to the next chapter where we look in detail at debugging techniques.

10 Debugging

No matter how experienced you get at writing code, there will be times when the code you write doesn't work the way you expect. Isolating and fixing the errors is called *debugging* and is a fascinating mental challenge or a complete headache depending how you are feeling that day! In this chapter we will look at how you can use the many tools in Director to help with your debugging, we look at many of the more common errors and we look at techniques for isolating the problems and then fixing them.

The general approach

Every debugging exercise is going to be different. Unfortunately I can't offer a step-by-step cure-all. What I can offer, however, are some general guidelines and lots of advice on how to use Director's many debugging tools to the best effect. But there is one golden rule: always save several copies of your movie. One method is to name the movie with consecutive numbers, which is the technique I prefer; other developers have their own preferences. The first save of a movie will be 'mymovie01.dir'; when saving after a half-day session, save the movie as 'mymovie02.dir'. In this way you can always step back to a point where your movie worked.

What's going wrong?

One of the most difficult aspects of debugging is identifying where the problem lies. Lingo code can be added in many places: a frame script, a movie script or a behavior. As your movie runs at up to 50 frames per second, how do you analyze whether the errors exhibited stem from one of several potential scripts?

The first step is to stop all code running. For each frame and behavior script that has an enter or exit frame handler, place a return on the first line.

```
1  on exitFrame me
2      return
3      -- More Lingo
4  -- ...
5  end
```

Listing 10.1

This ensures that any errors attributed to a specific script are eliminated. Run the movie; even though the movie will not do anything you should have no other errors. Then one-by-one comment out the return line using the comment symbol '--'. In this way you should at least be able to focus on the script that is causing the errors.

Errors attributed to syntax

Like all programming languages, Lingo is very dependent on syntax. In everyday language we can say the same thing in many different ways; in Lingo there is usually only one correct way. Fortunately if you have got this wrong then Director will tell you. If you save the movie then any script errors will result in a message box offering a button to ‘Show Errors’; clicking this opens the ‘Script error:’ dialog. This gives an error message, the line where the error occurred and a ‘?’ symbol indicating where on the line the error was found. Figure 10.1 shows the error found from the Lingo in Listing 10.2.

```
1 on exitFrame me
2     x = y + 6
3 end
```

Listing 10.2

This error is fixed by making the variable ‘y’ a property of the script.

```
1 property y
2
3 on exitFrame me
4     x = y + 6
5 end
```

Listing 10.3

Syntax errors are probably the most common bug in any programming language and Lingo is no exception. Always check for

- Lingo key word spelling – the word will go green. Typing errors are easy to make.
- Variable names – ensure that you haven’t accidentally written *cdeNum*, when you meant *codeNum*. Such mistakes are again very easy to make.
- Variables with the same name – it is easy to create a global variable and then accidentally have a property with the same name. The global variable is initialized but the property supersedes it. ‘Examples/Chapter10/Debug02.dir’ shows this in action. Fixing the problem is simply a matter of changing the property to a global.
- The symbol for combining elements of a string in Lingo is ‘&’ not ‘+’ as in several other scripting languages, including JavaScript.

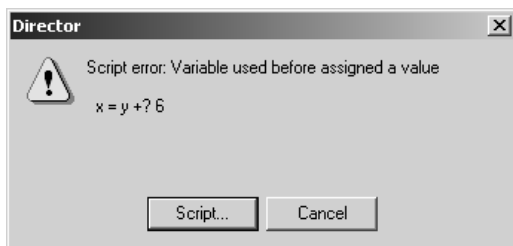


Figure 10.1 ‘Script error:’ dialog box

Name	Value
[-] sprite(1)	
- spriteNum	1
- member	(member 3 of castLib 1)
- startFrame	1
- endFrame	28
- editable	0
- visible	1
+ color	rgb(0, 0, 0)
+ bgColor	rgb(255, 255, 255)
- blend	100
- ink	0
+ loc	point(229,172)
- width	40
- height	37
- rotation	0.0000
- skew	0.0000
- flipH	0
- flipV	0
- cursor	0
[-] scriptInstanceList	
[-] [1]	<offspring "" 2 2288304>
- spriteNum	1
- mycount	270

Figure 10.2 ‘Object Inspector’ window for ‘Examples/Chapter10/Debug03.dir’

If the Script window closes without an error message, the script may still contain a syntax bug but some of the more obvious problems have been eliminated.

Using the Object Inspector

If the problems relate to a particular sprite on the stage then one of the first things to test is the internal state of the sprite using the Object Inspector.

Select ‘Window/Object Inspector’ to make sure the pane is visible. Then click and drag the sprite from the ‘Score’ window, not the ‘Stage’ window; make sure you use the Score window. Drop the drag onto the ‘Object Inspector’. It then adds a great deal of data to the pane. If you open ‘Examples/Chapter10/Debug03.dir’ then you should get the pane shown in Figure 10.2.

Lots of information is visible, about the sprite and, if the sprite has a behavior, about the scripts attached. If the movie is running then the ‘scripInstanceList’ can be expanded using the ‘+’ button and the behavior of the internal properties is constantly updated. Sometimes a difficult-to-track error can be spotted this way by carefully reading the contents of the internal properties.

Using the ‘Message’ window

One problem with the ‘Object Inspector’ window is that the data only lasts while the frame is running. Often you would like a dump-out of how the data looked at one instance and then again a moment later. Here we use the ‘Message’ window; ‘Window/Message’ opens the window or brings it to the top if it is already running. The top section of the Message window is the ‘Input pane’ and the lower part is the ‘Output pane’. If the Output pane is minimized, using the central arrow button, then the Input pane receives all the output.

Try typing Lingo into the ‘Input pane’. Press ‘Enter’; if you use a ‘put’ instruction then you will see the results in the ‘Output pane’. Figure 10.3 shows a short example. Once the line is in the Input pane you can run it again by positioning the cursor at the end of the appropriate line and pressing Enter.

The buttons on the ‘Message’ window tool bar, running from left to right, are shown in Table 10.1.

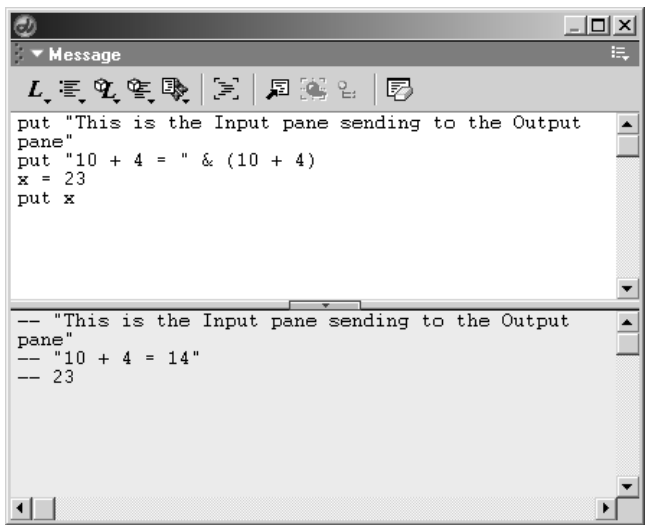












Figure 10.3 *The ‘Message’ window used for direct Lingo entry*

Table 10.1 *Buttons in the ‘Message’ window*

	Alphabetical Lingo	Pop-up menu to select Lingo key words by name
	Categorized Lingo	Pop-up menu to select Lingo key words by type
	Alphabetical 3D Lingo	Pop-up menu to select 3D Lingo key words by name
	Categorized 3D Lingo	Pop-up menu to select 3D Lingo key words by type
	Scripting Xtras	Pop-up menu to select Xtras key words by name (we will look into Xtras later in the book)
	Trace	Simple button to turn ‘Trace’ on or off
	Go to Handler	If a handler is highlighted in the ‘Input pane’ then this button will jump to the script for that handler
	Watch Expression	If the ‘Debug’ window is open and a variable is highlighted in the ‘Input pane’ then the value will be added to the ‘Watch’ window. More on this later
	Inspect Object	If a variable is highlighted in the ‘Input pane’ then clicking this button will add it to the ‘Object Inspector’ pane
	Clear	Clears the ‘Output pane’

The 'Trace' button in the 'Message' window is very useful for determining the order of Lingo statements. It places the statement in the Message window 'Output pane' when it is activated.

```

1 == Script: (member 1 of castLib 1) Handler: beginSprite
2 --> mycount = 0
3 == mycount = 0
4 --> end
5 == Script: (member 2 of castLib 1) Handler: startMovie
6 --> put "start"
7 -- "start"
8 --> end
9 == Script: (member 1 of castLib 1) Handler: exitFrame
10 --> incrementCount
11 == Script: (member 1 of castLib 1) Handler: incrementCount
12 --> mycount = mycount + 1
13 == mycount = 1
14 --> end
15 == Script: (member 1 of castLib 1) Handler: exitFrame
16 --> incrementCount
17 == Script: (member 1 of castLib 1) Handler: incrementCount
18 --> mycount = mycount + 1
19 == mycount = 2
20 --> end

```

Listing 10.4

Listing 10.4 shows the results of running 'Examples/Chapter10/Debug04.dir'. A line that starts with '==' shows the movie location. A line that starts with '-->' shows the Lingo line being executed. A line that starts with '--' is the result of a put operation. By tracing through the lines it is sometimes obvious where an error occurs. This technique can add a huge amount of data to the 'Message' window, which can be cleared using the 'Clear' button. Sometimes it is necessary to save the output to a file; this is achieved by setting the value of the 'traceLogFile'. 'Examples/Chapter10/Debug05.dir' shows how this is done. The important thing is to make sure that you close the file by setting the value to "", as the movie closes, otherwise you will not see the results in the file.

```

1 property mycount
2
3 on beginSprite
4   mycount = 0
5   _movie.traceLogFile = _movie.path & "dump.txt"
6 end
7
8 on endSprite
9   _movie.traceLogFile = ""

```

```

10 end
11
12 on exitFrame
13   incrementCount
14 end
15
16 on incrementCount
17   mycount = mycount + 1
18   put "mycount = " & mycount
19 end

```

Listing 10.5

Listing 10.5 shows how to create the file at line 5 and then close it down on line 9. Any text that gets sent to the ‘Message’ window also appears in this text file.

More debugging tips

- For every handler and function use a ‘put’ on the first line that places information about the function and any parameters to the ‘Message’ window or a log file.
- Be systematic; don’t make lots of changes at once.
- If a change does not fix a problem then change it back.
- Don’t simply copy and paste, as this may introduce variable naming errors or program state errors that caused the function to work in one place and not in another.

Using the Debugger window

When all else fails, using breakpoints is the only option (see Figure 10.4). Open the ‘Script’ window for ‘Examples/Chapter10/Debug06.dir’ and display the script ‘Problems’ (right click it from the ‘Cast’ window and select ‘Cast member script...’ or select ‘Window/Script’ and use the forward and back buttons to navigate to Problems). The purpose of this simple program is to slide the black box from left to right and back. The problem is it goes off the screen at the right, never to return. We want to find out why, using the ‘Debugger’ window. Click on the blue bar with the line number 9 in it. You

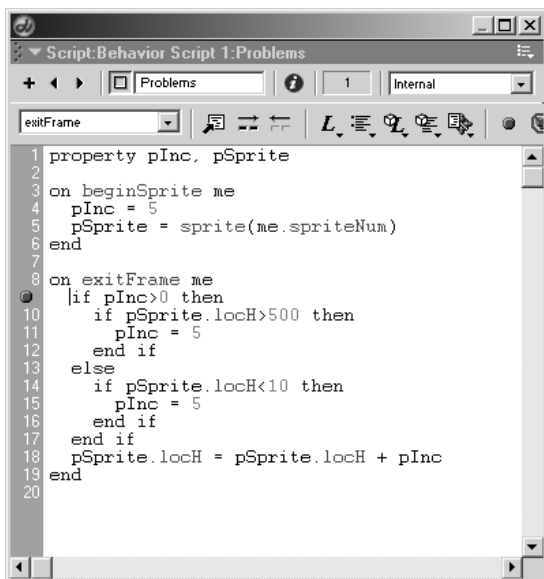


Figure 10.4 A breakpoint

should see a red circle appear. This circle is the visual indication of a breakpoint. When the program runs it will stop at this point and display the ‘Debugger’ window.

Try running the program and you should then see the window shown in Figure 10.5.

The ‘Debugger’ window provides a wealth of information for the developer but can be a little intimidating at first. The left pane has a series of panels. The top panel is the ‘Call Stack’. When a program runs one handler may call another which in turn may call a third.

```

1  on exitFrame me
2    iterate 5
3  end
4
5  on iterate n
6    if n>0 then iterate (n-1)
7  end

```

Listing 10.6

Listing 10.6 shows a very simple movie ‘Examples/Chapter10/Debug07.dir’; this movie uses a technique called iteration, where a function calls itself. If you place a breakpoint on line

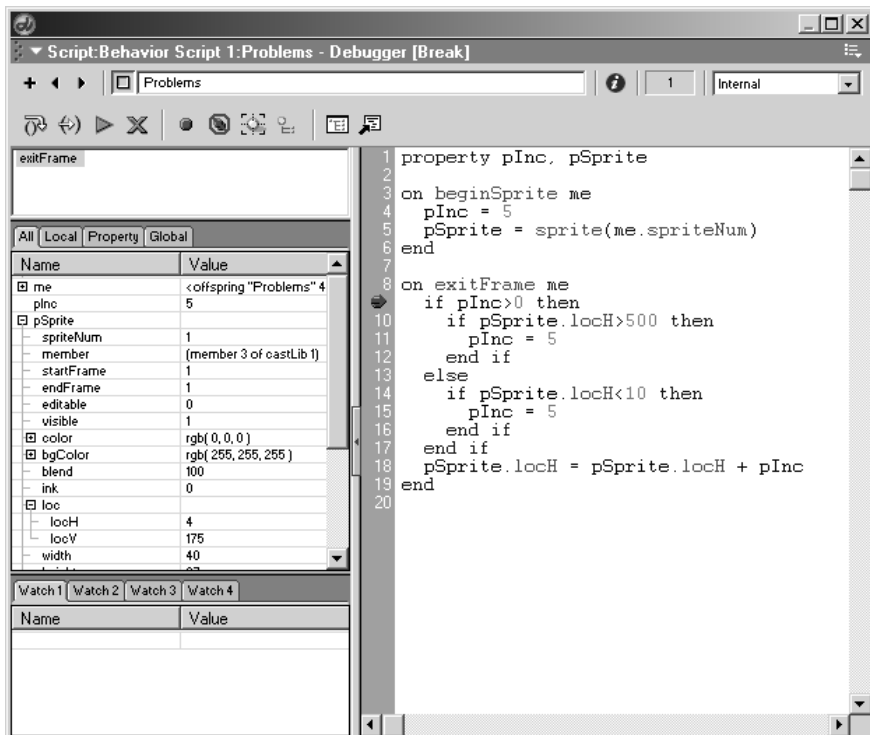


Figure 10.5 The ‘Debugger’ window

2 by clicking in the blue area of the 'Script' window, then by running the movie you can find out what happens. The movie stops showing the debugger window. If you now press 'F8', then you step into the script, which takes you to the function 'iterate' with 'n' equal to 5. Line 6 in the function tests the value of n; because it is greater than zero the function is called again, this time with n equal to n minus 1. This repeats until n is equal to zero. At this stage the 'Call Stack', as shown in Figure 10.6, has six versions of the iterate function together with the original calling handler 'exitFrame'. Each one of the instances of the function will have a different set of local variables. By clicking on each of the function names in the Call Stack pane you can analyze the value of the variables.

Try stepping into the movie until all six versions of iterate appear in the 'Call Stack'; now click on each one in turn and look at the value of 'n' displayed under the 'All' tab of the middle left pane. The value changes even though the movie is parked. Lingo is handling lots of different functions at once in this example.

The 'Variable' pane allows you to filter the variables. In this simple example it is fine to see all the variables at once, but as your movies become more complex you will want to filter the data.

- 'All' tab – displays both global and local variables available to the current handler/function.
- 'Local' tab – displays just local variables available to the current handler/function.
- 'Property' tab – displays just property variables available to the current handler/function.
- 'Global' tab – displays just global variables available to the current handler/function.

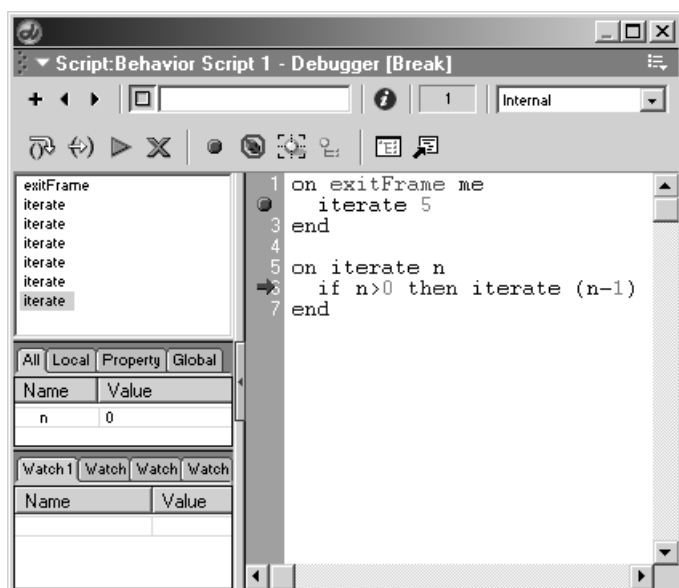


Figure 10.6 Running 'Examples/Chapter10/Debug07.dir'

'Examples/Chapter10/Debug08.dir' is a movie that includes global, property and local variables. Full details are given in Listing 10.7.

```

1  --Movie Script
2  global gCount
3
4  on prepareMovie
5    gCount = 1000
6  end
7  --End Movie Script
8
9  --Frame Script
10 property pCount
11 global gCount
12
13 on beginSprite
14   pCount = 0
15 end
16
17 on exitFrame me
18   iterate 5
19 end
20
21 on iterate n
22   pCount = pCount + n
23   localCount = pCount
24   if n > 0 then iterate (n - 1)
25 end
26 --End Frame Script

```

Listing 10.7

The 'Movie Script' initializes a global variable 'gCount' to 1000; the 'Frame Script' initializes the property 'pCount' to zero. Then the 'exitFrame' handler calls the iterate function. The iterate function amends the value of pCount in line 22 and a local variable 'localCount' is created and set to the current value of pCount. Essentially this is a do-nothing movie, the purpose is simply to illustrate the information that can be derived from the various tabs of the variables pane. By stepping through the script you can examine the value of the global, property and local variables as the movie executes each line of Lingo.

Often you are presented with far more information than you either want or need. When this is the case the bottom pane of the 'Debugger' window comes to the fore. This allows you to choose specifically which variables to view. This pane is called the 'Watch' pane and allows you to store four different sets of Watch variables. Figure 10.7 shows the Watch pane being used.

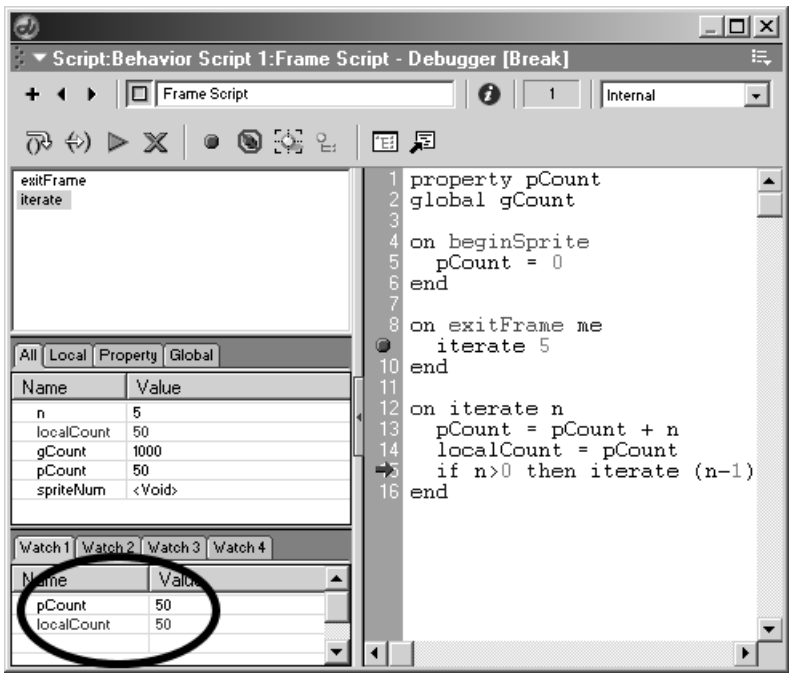


Figure 10.7 Using the ‘Watch’ pane

To add a variable to the Watch pane simply highlight it in the ‘Script’ pane, right click and select ‘Watch Expression’. To remove it from a Watch pane, select the variable in the Watch pane and press ‘Delete’.

Now that you are familiar with the basic operation of the ‘Debugger’ window, let us go back to the earlier problem of the box that stubbornly refused to slide back and forth, ‘Examples/Chapter10/Debug06.dir’. Recall that we added a breakpoint and if you run the movie it stops parked at line 9. The first time you step through the code, ‘pSprite.locH’ has a value of 4. At line 18 this goes up by the value of ‘pInc’, which is set to 5 by the ‘beginSprite’ handler. If just 5 is added each time the ‘exitFrame’ handler is executed, then the test at line 10

```
if pSprite.locH > 500 then
```

will not evaluate to true until 100 frames have been executed. That is going to take an enormous amount of stepping through! As we are interested in what happens when this test does evaluate to true, we should move the breakpoint from line 9 to line 11, inside the test. Try this now or open ‘Examples/Chapter10/Debug09.dir’ where it is already set up. This time the movie plays and the black box moves across the screen until it arrives at the right-hand side. At this point the variable pSprite.locH is more than 500, so the code inside the ‘if’ statement executes; because this code has a breakpoint set, Lingo breaks and the Debugger window opens. Notice that the value for pSprite.locH is now 504. At this point the line should set the value of pInc to –5 not 5. You can edit the line in the ‘Script’ pane, but remember to move the cursor to a new line so that Director knows to use your amended script. Press the green arrow button or ‘Run Script’ or press ‘F5’ to

see how your changes have affected the movie. Using this method you should see the black box moving left and right as the movie runs.

Often the code at breakpoints is not as obvious but the methodology remains the same; break where you think there is a problem and examine the value of key variables. Any that are not as expected can help you determine where a problem lies.

Using ‘debugPlaybackEnabled’

Another useful technique when debugging is to use

```
player.debugPlaybackEnabled = true
```

Using this simple code in your ‘prepareMovie’ handler will ensure that a message window is displayed even when the movie is shown in a browser. This allows you to examine the ‘puts’ or ‘traces’ that you place in your scripts even when embedded into an HTML page.

A ‘prepareMovie’ handler can be created as follows.

- Create a movie script by opening the script window and pressing the ‘+’ button.
- Open the ‘Property Inspector’ for the script.
- Click the ‘Script’ tab.
- Set the type to ‘Movie’
- Create a handler in the script as shown in Listing 10.8.

```
on prepareMovie
    player.debugPlaybackEnabled = true
end
```

Listing 10.8

Summary

Debugging is something that is learnt by experience. Each project will require a different technique in order to detect where the code is in error. In this chapter we have looked at some tips and techniques that will help you get started when debugging your code.

11 Integrating with Flash

Before we head off to the sunny pastures of creating our own games in Director there is one very important technique that is worth learning. So far every one of the games we have produced using Director has had some kind of Flash interface; check out Appendix B for a series of links. One of the key benefits of Director is that many other types of media can be incorporated, Flash being one of these. In this chapter we will look at how to communicate between these two development platforms.

Flash or Director?

Because you are reading this book, you obviously feel that Director has some benefits that make learning how to use the tool worthwhile. Some Flash developers are more sceptical, feeling that there is nothing in Director that cannot be done in Flash; some Director developers feel the opposite. The truth, as always, is somewhere in the middle. Flash makes creating a custom interface very easy, but it's not great with bitmaps and it cannot handle real-time 3D. Director is good for combining media created in many development environments, video, 3D and Flash, and it is very good at handling bitmaps fast. If you are creating a game that makes extensive use of bitmaps then Director is likely to be a better choice than Flash. Creating a custom interface in Director can often be more difficult than in Flash, which has multiple timelines that make simple things, like revealing a button, much easier to develop. But Flash cannot embed another technology readily, although Flash MX 2004 Professional has introduced the ability to add an extension. Later in this book we will see how you can use Director to create real-time 3D games; this single feature alone makes the game developer want to learn to use Director. Nevertheless, Flash is terrific for many things and so learning how to incorporate it into your movie is well worth the trouble.

Preparing your Flash project

The example we are going to use in this chapter is shown in Figure 11.1.

The lower quarter of this screen is a Flash sprite created using Flash MX 2004 Professional. The Flash sprite is placed on the score in channel 1. A shockwave 3D sprite appears in channel 2. The aim of this chapter is to show how a communication is achieved between the Flash sprite that incorporates usable buttons and the 3D sprite.

A brief introduction to Flash for Director developers

Many readers of this book will have only a vague knowledge of Flash and so many may choose to avoid using it. This would be a mistake as it is a great tool for developing a user interface (UI). Flash uses a timeline and layers, which are the exact parallel to the score in Director. Navigating around

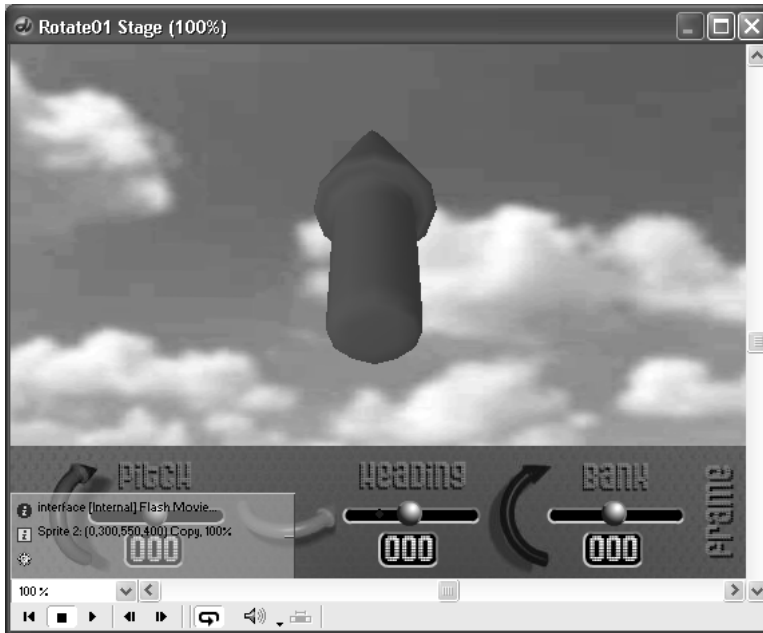


Figure 11.1 *'Examples/Chapter11/Rotate01.dir movie'*

the timeline can be achieved by using 'ActionScript', Flash's built-in programming language, which is very similar to JavaScript. If you have followed the JavaScript examples presented in this section of the book then you are well ahead in your attempt to learn ActionScripting. The elements that you can place on the timeline can be

- shapes (which are easily drawn using the toolbox in Flash)
- bitmaps (which can be imported)
- graphic symbols
- buttons
- 'MovieClips'.

In a UI you will almost certainly need to know how to use buttons. The technique is very simple. Flash uses multiple timelines. Suppose you create the design of a button that you are happy with. Then to turn this into a full functioning button you highlight the graphics and press 'F8'. This will allow you to turn the graphic shapes into a button; first you will see the dialog box that appears in Figure 11.2.

By selecting the 'Button' option you are telling Flash that the shapes will ultimately be used as a button. Each symbol type is simply a new timeline. In the timeline you are free to add graphics and layers just as you can on the root timeline. This is one of the most useful features of Flash. A button is a very specialized symbol where each of the first four frames of the timeline have a different and unique purpose. Frame 1 is the image you will see when the button is not active, frame 2 is the image you will see when the mouse is over the button, frame 3 is the image you will see when the

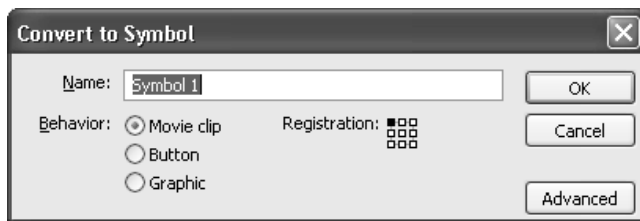


Figure 11.2 *Creating a symbol in Flash MX 2004*

mouse has clicked the button and frame 4 defines the hit area for the button. Each of the frames is optional; you can create an invisible button simply by defining just the hit area.

Symbols are stored in the library and can be used like cast members in Director. Once you have placed a button on the timeline you use ‘ActionScript’ to make it trigger events. To add some ActionScript, click on the button, open the ‘Action’ window by pressing ‘F9’ and type in the code.

```
1 on (release){
2     getURL("event:triggerDirector");
3 }
```

Listing 11.1

As you can see from Listing 11.1, ActionScript uses JavaScript syntax. A function’s parameters are passed between parentheses and the function’s statements are contained between curly braces. Each statement is separated by a semi-colon. A button can have several other options other than ‘release’ as a parameter. When triggering Director we need to use the syntax of line 2 in Listing 11.1. Where this example uses ‘triggerDirector’ after ‘event:’ any name can be used.

Space is limited so I advise you to check out some of the books in the bibliography to learn how to really get to grips with Flash. But hopefully this short introduction will help your understanding when looking at the example shown in this chapter.

Preparing your Director project

Now would be a good time to open ‘Examples/Chapter11/rotate01.dir’. When developing a Director movie that incorporates a Flash interface you will often need to edit the Flash interface while developing in Director. It is essential that you have a copy of Flash on your development machine, ideally the latest version Flash MX 2004 and better still the Professional version. The link between Director and Flash from Macromedia is now robust and easy to use. The Flash component that you add to your Director movie is the published form of Flash, the swf file. To publish your Flash movie check the publish settings options by opening the ‘Publish Settings’ panel, accessed by selecting ‘File>Publish Settings...’.

Notice from Figure 11.3 that one of the publish options is ‘Flash (.swf)’. It is this file that you will import into Director. The file that Flash uses to store the editable project is the fla file. To prepare your Director movie for convenient editing you must import the swf file and then tell Director where to find the Flash project (fla) file. To do this you open the ‘Property Inspector’ in

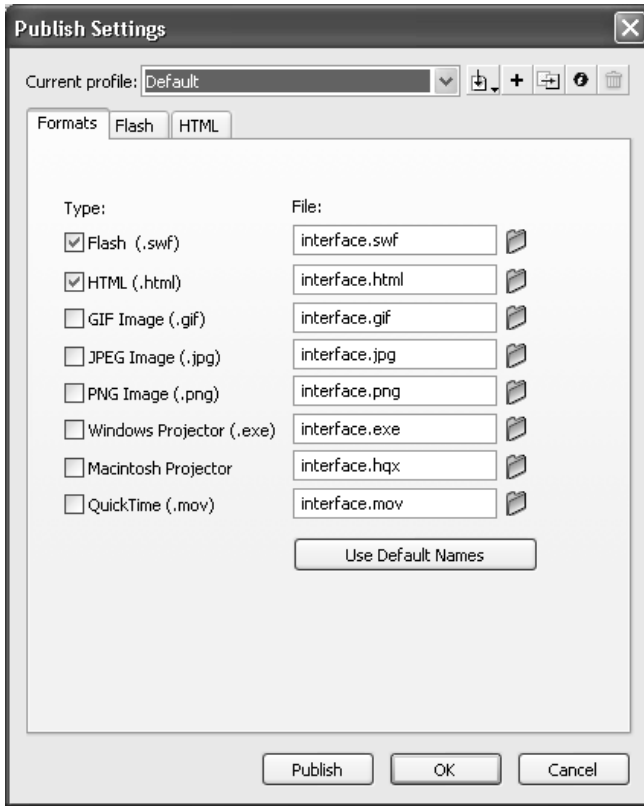


Figure 11.3 'Publish Settings' panel in Flash MX 2004

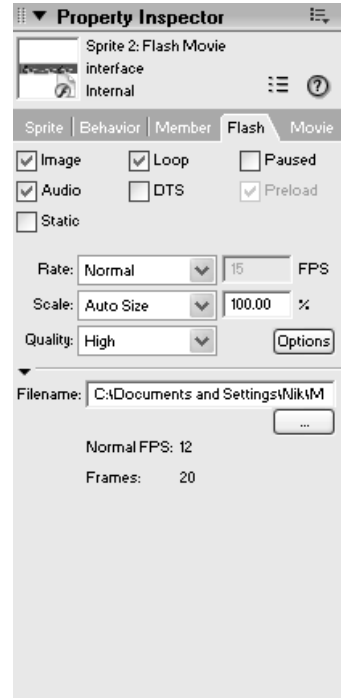


Figure 11.4 Setting the filename for a Flash sprite

Director when you have a Flash sprite highlighted. Notice that there is a Flash tab. Clicking this shows the Director panel you can see in Figure 11.4.

Click the '...' button underneath 'Filename:' and navigate to the fla that was used to create the imported swf file. Once this is set you will be able to edit the Flash document using Flash by double clicking on the Flash sprite on the stage. This opens Flash on your computer with the fla ready to edit. Once you have completed your amendments to the Flash sprite, you finish the editing process by clicking the 'Done' button in Flash; this will return you to Director with a newly compiled swf in your cast. As you develop movies that have a lot of interaction between Flash and Director you will regularly switch between the two. To avoid Flash initializing every time, a procedure that takes several seconds, open a blank document in Flash, which will stay open after you click the Done button and so the switch will be much quicker.

Accessing Flash variables and objects

The easiest way to manipulate variables in the Flash sprite from Director is to ensure that the variables are at the `_root` of the Flash movie. Flash can embed a movie clip symbol that is effectively a complete movie in its own right, with variables that are within its own scope. If you create a Flash

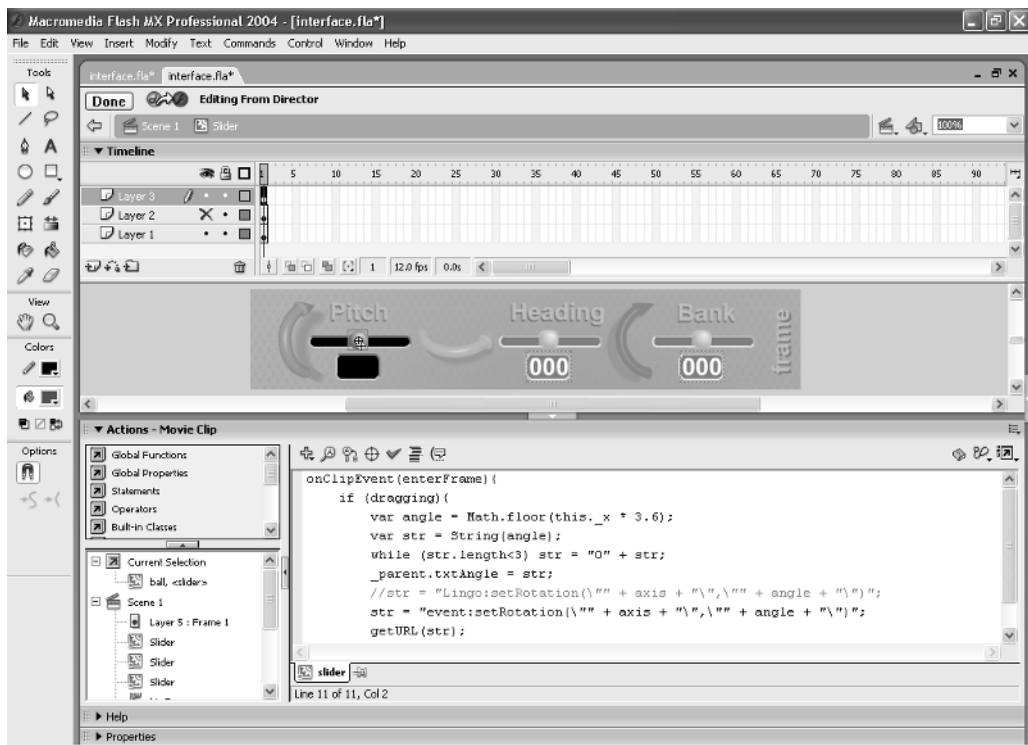


Figure 11.5 *Editing in Flash from Director MX 2004*

movie that embeds a movie clip with an instance name of 'myclip', then to access the variables in this clip from Flash, you would use 'myclip.myvariable', where myvariable is replaced by whatever name you have used for your variable. It is possible to use the same techniques when using Director. First we need to know the difference between accessing a Flash variable by value or by reference. If we access a Flash variable by value then we are exchanging the value using a string. Every value we receive will initially be a string value. Using the conversion tools in Director this value can easily be converted to a numeric value but remember that it is originally a string.

```
str = sprite(1).getVariable("myvariable")
```

First we define the sprite channel where the Flash is stored, then we use the method, 'getVariable' and finally we pass the name of the variable. But what if the variable we require is inside a movie clip in Flash? Then we use the alternative version and get a reference to the Flash object.

```
mc = sprite(1).getVariable("myobject", false)
```

Notice that we are using a second parameter. Here the value of the parameter is set to 'false'. This informs Flash that the return value of the getVariable method should be a reference to a Flash object. An object can be a movie clip, an array or any other Flash object. Suppose that the return

value is a movie clip, then the variables and properties of this object are retrieved using the following syntax:

```
--If the Movie Clip is located at 100.3, 64.7 then
put "(" & mc._x & ", " & mc._y & ")"
--will display (100.3, 64.7)
--To access custom variables use
spd = mc.speed
```

You can use Flash arrays using

```
pArray = sprite(2).getVariable("axisNames", false)
put pArray[0] && pArray[1] && pArray[2]
```

Flash arrays work similarly to Director lists, but remember that the first item in the array is at index zero not 1.

Once you have a Flash reference you can use the reference to get and set values. If you are working with the value option then you can set the value directly using

```
--When the variable is a string value just use the variable ↓
name
sprite(1).setVariable("variableName", stringvariable))
--When the variable is numeric convert it before passing it
sprite(1).setVariable("variableName", String ↓
(numericvariable))
```

Notice that the value passed is always a string, so if the value being passed is numeric, then remember to use the second of the above two syntax options where the variable is converted to a string form.

Controlling Flash from Director

There are two approaches to controlling Flash from within Director. The first option is to control the Flash sprite. Assuming that your Flash sprite is on channel 1 you can use this method to set the frame of the `_root` timeline:

```
sprite(1).gotoFrame(x)
```

If the timeline you wish to manipulate is not at the root, then you can use the Flash 4 style of

```
1 sprite(1).tellTarget("myMovieClip")
2 sprite(1).gotoFrame(x)
3 sprite(1).endTellTarget()
```

Listing 11.2

Referring to Listing 11.2, first you inform Flash that subsequent commands target the name passed as a parameter to the 'tellTarget' function. Then you can control the nested timeline in the same way as controlling the `_root`. When finished it is good form to send an 'endTellTarget' as shown in line 3 of Listing 11.2, this ensures that subsequent commands are sent to the `_root`, the default

behavior. You can alter nested variables using the same methodology. If you are using Flash movie clip objects then it is easier to use Flash style coding.

```
1 pControlMC = sprite(1).getVariable("myMovieClip", false)
2 pControl.gotoAndStop(pControlMC._currentframe + 1)
```

Listing 11.3

First you get a reference to the movie clip and then you have access to all the usual Flash properties including the current frame number. To move the playback head for the movie clip simply use the Flash function ‘gotoAndStop’ or ‘gotoAndPlay’. Using references makes the control of complex Flash movies as easy and convenient as working directly in Flash.

Controlling Director from Flash

If you are using a Flash movie as an interface then you will want Flash buttons to trigger Director events. You can do this quite easily. First set up your buttons to send a ‘getURL’ command.

```
1 on (press) {
2     getURL("event:triggerDirector");
3 }
```

Listing 11.4

Listing 11.4 shows how a button would be coded in Flash. The getURL command takes a single parameter, which is the string ‘ “event:xx” ’ where xx can be any string value that contains function-naming characters. Now in Director make sure there is a movie script ready to receive this command. It must be a movie script; frame and behavior scripts are unsuitable.

```
1 on triggerDirector
2     --Do whatever
3 end
```

Listing 11.5

Suppose that you want to send parameters to the function. Then you must use

```
1 on (press) {
2     getURL("event:triggerDirector, "+ arg1 + ", " + arg2);
3 }
```

Listing 11.6

Notice that we still pass a single string and that the function name is immediately followed by a comma. Each parameter has a comma and a space preceding it.

```
1 on triggerDirector me, arg1, arg2
2     --Do whatever
3 end
```

Listing 11.7

Make sure that you use the 'me' key word before any arguments in the receiving Director function. You can also pass data between the Flash sprite and Director by accessing the current variable values from the Flash sprite inside the Director function.

```
1 on triggerDirector me, arg1, arg2
2     n = sprite(1).getVariable("extravariabe")
3 end
```

Listing 11.8

Sending Lingo from Flash

You can also use Flash to send direct Lingo commands. This means there does not have to be a receiving function in the Director movie.

```
1 on (press){
2     getURL("lingo:put \"Hello from Flash\"");
3 }
```

Listing 11.9

Summary

Flash and Director communication is easy to use and versatile; in this chapter we looked at the principal techniques that are used to manipulate and control Flash from Director and vice versa. If you want to have an interface that is elegant, dynamic and small in terms of file size then it is well worth learning how to use Flash and how it is incorporated into your Director movies.

This page intentionally left blank

Section 3

Putting it into practice

You know your stuff so how about using this knowledge to make some games?

This page intentionally left blank

12 Kids' stuff

Director has been the most popular authoring environment for multimedia CD-ROMs for several years. There have, in this time, been countless CD-ROMs aimed at a very young market, anything from talking books to print activities and simple games. In this chapter we are going to look at the strategies involved in creating this type of movie. In many ways they are a simple introduction to game scripting because in general they are much simpler to set up than a full-on game. We will look at a couple of examples that come from our back catalogue. Most of these date back to a time when we all had to work with 256 color displays. Thankfully for most of us those days are behind us. Full color displays give a much better image and allow us to handle just about any images without having to worry about palettes.

A simple talking book with animated events

'Examples/Chapter12/Ronnie.dir' is a single page from a 16-page talking book. Figure 12.1 shows the main action. As the movie starts we hear a narrator reading the words, which are highlighted as the words are spoken. Once the text has been read, the action parks on a looping animation of the two key characters. At this stage, the child can click on the green ball to have the text read again or

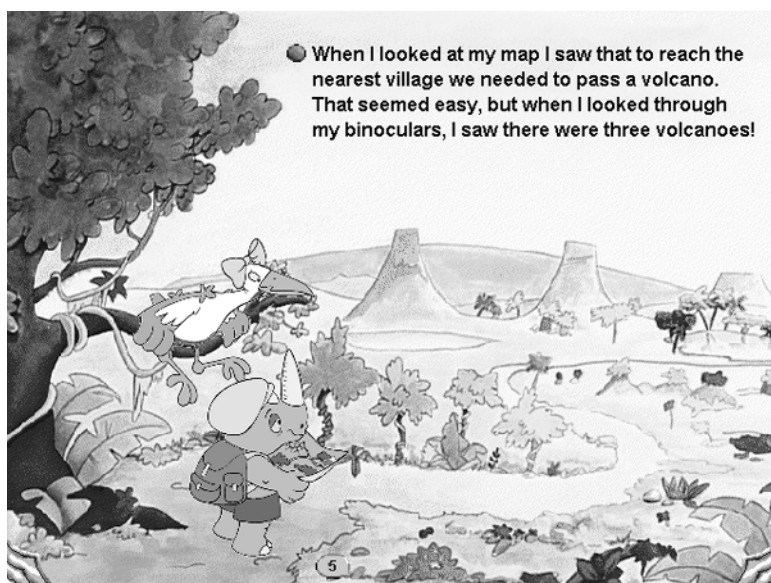


Figure 12.1 'Examples/Chapter12/Ronnie.dir'

click an individual word to have that single word spoken. As you move the cursor over the page, it changes to a different shape when we have a clickable event. There are eight ‘clickables’ on the page:

- juggling giraffe
- monkey in the tree
- screaming bush
- spaceship
- snake into rucksack
- the map flying away
- bats flying out
- a joke.

Each of the animation events is handled by a separate section of the score, so the only thing the developer has to worry about is to send the playback to the appropriate place in response to a ‘mouseUp’ event. This is handled by an invisible button. A shape is added to the stage using the ‘Tools’ palette. Then a script is added to this shape in the ‘Cast’ window and finally the ink for the shape is set to ‘Background Transparent’ using the Property Inspector for the sprite. The script has the text seen in Listing 12.1.

```
1 on mouseUp
2   go to "Bush"
3 end
```

Listing 12.1

The joke is handled simply by starting a random sound file.

```
1 on mouseUp
2   sound(1).playFile("sounds\joke" & random(15) & ".aif")
3   repeat while sound(1).isBusy()
4     end repeat
5 end
```

Listing 12.2

Perhaps the most complex job in this activity is synchronizing the sound and highlighting and playing the individual words once the text has been initially read. To do this we create a list of four lists, one for each line of the text on the page. Each entry in these lists has a property name, always prefixed with a ‘#’ symbol and a number indicating the horizontal position of the word from the top left corner of the block of text.

```
1 on startMovie
2   global gWords
3
4   gWords = [[], [], [], []]
```

```

5
6   gWords[1] = [ #When:40, #i:50, #looked:100, #at:125, #my:150, #map:190,
7                 #i:200, #saw:230, #that:260, #to:280, #reach:330, #the:360]
8   gWords[2] = [ #nearest:55, #village:105, #we:135, #needed:195, #to:205,
9                 #pass:255, #a:265, #volcano:335]
10  gWords[3] = [ #that:30, #seemed:95, #easy:135, #but:170, #when:215,
11                #i:225, #looked:280, #through:335]
12  gWords[4] = [ #my:25, #binocul:105, #i:120, #saw:155, #there:195,
13                #were:235, #three:280, #volcanos:330]
14
15 end

```

Listing 12.3

The sprite that handles the clicking of the text is in channel 10 of the score. The script is shown in Listing 12.4. Lines 4–8 take the position of the mouse and translate it into coordinates based on the top left corner of the text sprite. Line 9 takes the vertical value and converts this into the line that has been clicked. We now need to determine which exact word has been clicked. To do this we first create an offset value, which is the sum of the lines above; for line 1 this would be zero, for line 2 the total words on line 1, for line 3 the sum of the words in lines 1 and 2 and for line 4 the sum of the words in lines 1–3. Lines 19–25 test the horizontal location of the click against the values we originally stored in the 'gWords' array. If the value in the array is higher then the word clicked must be the current index. We use the method of a list 'getPropAt' to retrieve the property name at this index in line 21. We will use this to build a string value that is the file location for the sound that should be played. The cast number to display is derived by adding the sum of the words in the previous lines, which we have stored in the variable 'offset', to the current index in the repeat loop and then adding 123 to the result. The purpose of 123 is simple: this is the position of the first word in the cast. Subsequent words occupy consecutive cast number slots.

On line 27 we use the 'puppetSprite' method of the _movie object to tell Director that from now on the sprite in channel 10 is being updated using scripting rather than from the properties of the score. Then we position the sprite and set its cast number. At line 33 we use the property value to build a string that locates the sound file to play and in the following line this sound is started. Then we update the stage so we can see the results of the preceding Lingo instructions and wait until the sound has finished playing. Having done that, we restore the original cast member to the sprite in channel 10 and tell Director that the score is back in charge of this sprite.

```

1  on mouseUp
2    global gWords
3
4    wordH = _mouse.mouseH()
5    wordV = _mouse.mouseV()
6
7    wordH = wordH - sprite(2).locH
8    wordV = wordV - sprite(2).locV
9    whichLine = integer((wordV+10)/20.0)

```

```

10
11   offset = 0
12
13   if whichLine>1 then
14       repeat with index = 1 to (whichLine - 1)
15           offset = offset + gWords[index].count
16       end repeat
17   end if
18
19   repeat with index = 1 to gWords[whichLine].count
20       if (gWords[whichLine][index] > wordH) then
21           thisWord = gWords[whichLine].getPropAt(index)
22           wordCastNum = index + 123 + offset
23           exit repeat
24       end if
25   end repeat
26
27   _movie.puppetSprite(10, true)
28
29   sprite(10).locV = 25
30   sprite(10).locH = 254
31   sprite(10).castNum = wordCastNum
32
33   str = "sounds\" & thisWord & ".aif"
34   sound(1).playFile(str)
35
36   _movie.updateStage()
37
38   repeat while sound(1).isBusy()
39   end repeat
40
41   sprite(10).locV = 34
42   sprite(10).locH = 241
43   sprite(10).member = member("tBall")
44
45   _movie.puppetSprite(10, false)
46
47 end

```

Listing 12.4

Creating custom cursors

This example makes use of custom cursors. The cursor changes as the child moves the mouse around the screen and acts as a prompt to click at this location. Custom cursors use two images, the



Figure 12.2 *The image and mask of a custom cursor*

image you will see and a mask that defines the shape of the cursor. Each of these images must be a maximum of 16 pixels square and have a color depth of one bit. You can create them using Director's paint window or import them from your favorite paint program. Figure 12.2 shows the image and mask of one of the custom cursors.

Listing 12.5 shows how to use Lingo to assign one of these custom cursors to a sprite channel. You simply define a list. The value is the cast number of the image and the second is the cast number of the mask.

```

1  on exitFrame
2    global gChoice
3
4    repeat while soundbusy(1)
5    end repeat
6
7    if gChoice="Watch" then
8      go to "Watch"
9    else
10     sprite(2).cursor = [379, 380]
11     sprite(4).cursor = [374, 375]
12     sprite(6).cursor = [374, 375]
13     sprite(7).cursor = [374, 375]
14     sprite(8).cursor = [374, 375]
15     sprite(9).cursor = [378, 377]
16     sprite(10).cursor = [379, 380]
17     sprite(11).cursor = [374, 375]
18     sprite(12).cursor = [374, 375]
19     sprite(14).cursor = [381, 382]
20     sprite(15).cursor = [378, 377]
21     sprite(16).cursor = [378, 377]
22     sprite(17).cursor = [378, 377]
23     sound(1).playFile("sounds\music2.aif")

```

```

24     _movie.go("Main Cycle")
25 end if
26 end

```

Listing 12.5

The problem of fringing on imported cartoon artwork

The example above uses animation that was drawn on paper then scanned into the computer and colored. When importing into Director we have a problem due to fringing. This is caused by anti-aliasing in the image. There is a very simple solution to this problem, which works very well with cartoon artwork. If you intend to use an ink such as 'Background Transparent' to have an irregular surround to your sprite, then instead of having a white background for the sprite use a very dark gray. As long as the color is not black and your cartoon character has a black outline then the anti-aliasing will blend to a very dark color. When Director uses Background Transparent it takes a single color value and uses this for transparency. The anti-aliased edge of a character will have a series of colors that blend into the black line. None of these colors are the single color of the transparency so Director will display them. If you place the character over a light background then the effect of this fringing is very noticeable. If the anti-aliasing is from black to a very dark gray, then the actual dark gray is set to invisible but the remaining pixels are still very close to the black of the line so are not nearly as noticeable as when white is used. Figure 12.3 shows the fringing from a white background bitmap on the left and from a dark background bitmap on the right. Remember that you must set the correct background color for your sprite for Background Transparent to work when the choice is not white, the default option. Setting the color is simply a matter of using the 'Paint' window to get the correct color and then setting this in the 'Background Color' property using the 'Property Inspector' window for the sprite.

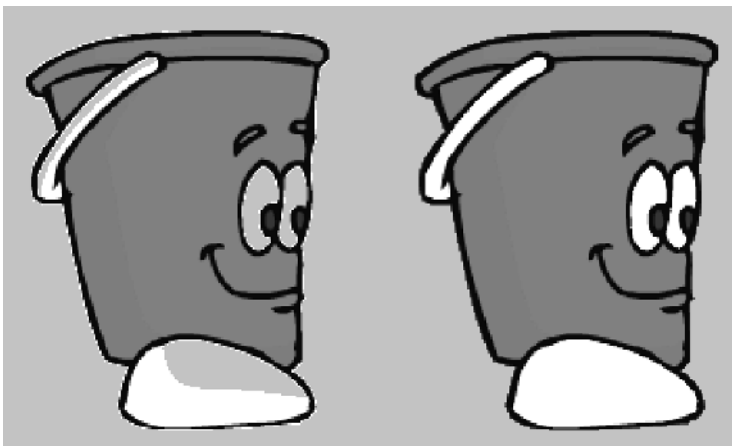


Figure 12.3 *Fringing with imported artwork*

A virtual train set

This second example is a little more complex, but not a lot. Try 'Examples/Chapter12/Trainset.dir' to see it in action. You are presented with an unmade train set and then you simply click the squares to cycle through different rail sections for that space. As you are doing this the train is shunting backwards and forwards. If you successfully complete a loop then the train will happily go round all day. The entire example relies on just 40 image tiles. The train is built into the tiles. You can see how the game looks in Figure 12.4 and the tiles that make up the game in Figure 12.5. So how is it done?

This example uses a frame loop script that is given in Listing 12.6. The purpose of the handler is to call the main function 'moveTrain' and then update the stage.

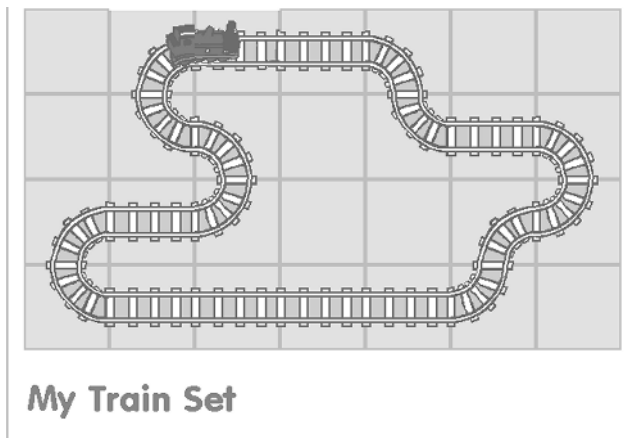


Figure 12.4 'Examples/Chapter12/trainset.dir'

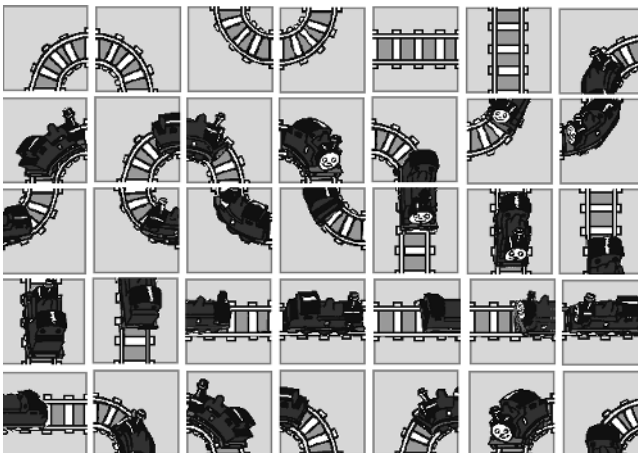


Figure 12.5 The images used to build the train set

```

1  on exitFrame
2    tile = _mouse.clickon()
3
4    if not sound(1).isBusy then
5      sound(1).playFile("Train.aif")
6    end if
7
8    moveTrain
9
10   _movie.updateStage()
11   _movie.go(_movie.frame)
12 end

```

Listing 12.6

The tiles are updated using cast member scripts for the tiles. Each tile has a different script but they are all essentially the same as the code in Listing 12.7. The listing shows the script for cast member 4. If this is on the stage at a particular location then the script would only be called if the value of ‘_mouse.clickOn’, the last sprite to be clicked, is the channel that this cast member is currently occupying. We can use this property to set the sprite’s cast number to point to the next in the library or any other we choose. Most of the scripts point to the next in the list, but the last loops this back to the start. As far as user interaction is concerned that is all we need to do. The more complex element is locating the train. The train can be exactly over a tile or half in one tile and half in another.

```

1  on mouseUp
2    sprite(_mouse.clickOn).castNum = 5
3  end

```

Listing 12.7

The function ‘moveTrain’ makes extensive use of lists that are initialized in the ‘startMovie’ handler shown in Listing 12.8.

```

1  global gPiece, gTrain1, gTrainDir, gLegalMoves
2  global gCurTile, gNextTile, gTrain2, gEnter, gExit
3  global gReverse, gDirections, gRevDir
4
5  on startMovie
6    sprite(2).castNum = 46
7
8    --global variable initialisation
9    gTrain1 = 12
10   gTrain2 = 12

```

```

11  gCurTile = 8
12  gNextTile = 8
13  gTrainDir = 1
14  gLegalMoves = [ 4,6,8,6,7,9,5,7,8,4,5,9]
15  gEnter = [ 37, 0, 0,10, 0, 0,13,34,19,40, 0, 0,\
16           0,16,43, 0,31, 0,28, 0, 0,22, 0,25]
17  gExit = [ 0,39,12, 0,36,15, 0, 0, 0, 0,42,21,\
18          18, 0, 0,45,33, 0,30, 0, 0,24, 0,27]
19  gDirections = [2,0,0,3,0,0,2,1,4,3,0,0,0,1,4,0,1,0,3,
20                0,0,2,0,4]
21  gRevDir = [0,3,2,0,2,1,0,0,0,0,4,3,4,0,0,1,3,0,1,0,0,4,0,2]
22  sound(2).volume = 150
23 end

```

Listing 12.8

The purpose of the variables is given in Table 12.1.

Listing 12.9 shows the 'moveTrain' function. This breaks readily into two sections; whether the global variables 'gTrain1' and 'gTrain2' are equal or not. If they are not equal then on the previous screen update, the train was across the boundary of two tiles. If they are equal then it was contained within a single tile. If it was across two tiles on the previous occasion then on this update it must be on a single tile, so we can call the function 'moveMidPos' without any further analysis. If it was on a single tile previously then it must be across two on this screen update. The challenge is to find

Table 12.1 *Global variables used in the movie 'Examples/Chapter12/Trainset.dir'*

Variable	Description
gTrain1	Holds the sprite channel of the track on which the rear of the train sits
gTrain2	Holds the sprite channel of the track on which the front of the train sits
gCurTile	The cast number of the track on which the rear of the train sits
gNextTile	The cast number of the track on which the front of the train sits
gTrainDir	The direction of the train (1 – left, 2 – down, 3 – right, 4 – up)
gLegalMoves	There are three possible exits from any cell given a specific train direction because it is not possible to exit using the entry direction. All the possible exits for each track type are given with this list. To select the appropriate value from the list use (gTrainDir-1)*3 and test each of the three consecutive values at this index
gEnter	Given the train direction we can use this list to determine which cast number to display for the front of the train
gExit	Given the train direction we can use this list to determine which cast number to display for the rear of the train
gDirections	Having moved to a new cell this allows you to update the current value of 'gTrainDir' based on the value of both gTrainDir and 'gNextTile'
gRevDir	Having found that the next cell is an illegal move this list allows you to update the current value of 'gTrainDir' based on the value of both gTrainDir and 'gNextTile' to force the train to reverse direction

which tile it is moving to and whether this space contains a legal entry point. A single cell has both an exit and entry point; if we are to allow the train to move then the exit point of a cell should connect with the entry point of the adjoining cell. The direction of the train's motion and consequent exit is stored in the variable 'gTrainDir'. Lines 5–14 of Listing 12.9 show the case of the train moving to the left. Line 8 tests to see if we are currently on the column furthest to the left; if we are then we can't move further in that direction so instead we need to calculate the reverse way out from the current location. The remainder of the moveTrain function handles the situations when the train is moving down, right or up. Each requires a slightly different test for the boundaries of the track and each has its own handler if a move is legal.

```

1  on moveTrain
2    if gTrain1<>gTrain2 then
3      moveMidPos
4    else
5      if gTrainDir=1 then
6        --moving left
7        --legal tiles are 4,6 and 8
8        if ((gTrain1-9) mod 7)=0 then
9          --at the edge can't move left
10         temp = gTrainDir+(gCurTile-4)*4
11         gTrainDir = gRevDir[temp]
12       else
13         moveLeft
14       end if
15     else if gTrainDir=2 then
16       --moving down
17       --legal tiles are 6,7 and 9
18       if gTrain1>29 then
19         --on bottom row can't move down
20         temp = gTrainDir+(gCurTile-4)*4
21         gTrainDir = gRevDir[ temp]
22       else
23         moveDown
24       end if
25     else if gTrainDir =3 then
26       --moving right
27       --legal tiles are 5,7 and 8
28       if ((gTrain1-9) mod 7)=6 then
29         --at the edge can't move right
30         temp = gTrainDir+(gCurTile-4)*4
31         gTrainDir = gRevDir[temp]
32       else
33         moveRight

```

```

34     end if
35     else if gTrainDir=4 then
36         --moving up
37         --legal tiles are 4,5 and 9
38         if gTrain1<16 then
39             --On top row can't move up
40             temp = gTrainDir+(gCurTile-4)*4
41             gTrainDir = gRevDir[ temp]
42         else
43             moveUp
44         end if--end move up
45     end if--end up section
46 end if--end check if mid pos
47 end

```

Listing 12.9

We will look in detail at the case when the train is to be painted into a single cell and the instance when the train is moving left. The remaining cases are all in the example movie and you can examine them while playing with the code. The visual representation of the train is held in the two sprite channels 37 and 38. These move around the stage so that the train always appears at the correct position on the track. When the train is directly over a single cell the location of the sprites will be held in the sprite in channel 38 which at the previous frame was the front of the train. To determine which cast member to show, we use the 'gEnter' list set in the 'startMovie' handler. Using the global variables 'gTrainDir' and 'gNextTile' we can get the index in the list that points to the correct cast member to display for this tile. 'gCurTile' holds the position of the rear of the train while gNextTile holds the position of the front of the train. When the train is entirely over a single cell both these values are equal, as are the values for 'gTrain1' and 'gTrain2'. Listing 12.10 shows how this case is handled and is completed by setting the direction of the train based on the current tile. This is stored in the global list 'gDirections'.

```

1  on moveMidPos
2      sprite(37).locH = sprite(38).locH
3      sprite(37).locV = sprite(38).locV
4      temp = gEnter[gTrainDir+(gNextTile-4)*4]
5      sprite(37).castNum = temp + 1
6      sprite(38).castNum = temp + 1
7      gCurTile = gNextTile
8      gTrain1 = gTrain2
9      temp = gTrainDir+(gNextTile-4)*4
10     gTrainDir = gDirections[temp]
11 end

```

Listing 12.10

Listing 12.11 gives the more complex case when the train needs to leave a cell and enter another, in this case by moving to the left. First we set the value of the variable 'legalMove' to false. The tile we need to check will be held in the sprite channel defined by 'gTrain-1'. Line 3 assigns the variable 'testTile' to the cast number of the sprite in this channel. Then we check the legal moves list; if it is OK then the value stored at the appropriate index in the legal move list will match the value for testTile. When this is the case we set the value of legalMove to true and the value of the global variable 'gNextTile' to the tested tile. We update the variable 'gTrain2' to point to the sprite channel tested and assign the back end of the train to the cast member of the rear section, stored in the list 'gExit'. Then we move the front section to match the location of the sprite channel we are moving to and set the front of the train image to that stored in the list 'gEnter'.

```

1  on moveLeft
2    legalmove = false
3    testTile = sprite(gTrain1-1).castNum
4    repeat with index = 1 to 3
5      temp = glegalMoves[(gTrainDir-1)*3+index]
6      if testTile= temp then
7        --legal move
8        legalmove = true
9        gNextTile = testTile
10       gTrain2 = gTrain1-1
11       temp = gExit[ gTrainDir+(gCurTile-4)*4 ]
12       sprite(37).castNum = temp
13       sprite(38).locH = sprite(gTrain2).locH
14       sprite(38).locV = sprite(gTrain2).locV
15       temp = gEnter[gTrainDir+(gNextTile-4)*4]
16       sprite(38).castNum = temp
17     end if
18   end repeat
19   --if no legalmove then set reverse Train
20   gReverse = not legalmove
21   if gReverse then
22     temp = gTrainDir+(gCurTile-4)*4
23     gTrainDir = gRevDir[temp]
24   end if
25 end

```

Listing 12.11

Legal move generators are often difficult to understand; if it is all rather vague then try working through the section again.

Summary

Director is the ideal tool for creating interactive activities for young children. There are many examples to look at; in this chapter we looked at a talking book and simple game activity but there are many more. Along the way we looked at how you can take a problem and present it in a way that a computer can understand, whether it is converting screen coordinates into cast member indices or testing the values in sprite channels to determine whether we can perform an action. In the next chapter we extend on the topic of legal move generators to look at some simple intelligence. If there are several legal moves then how do you decide the best from the current position? Read on.

13 Board games

There are many two-player board games that convert readily to being played by a solo player and a computer. Chess programming has particularly played a significant role in the history of computer programming. In this chapter I hope to introduce some of the ideas necessary to create a two-player strategy game that involves some programmed intelligence on the part of the computer.

Two-player board games

The simplest game is noughts and crosses. Even this game presents several challenges but with only a 3×3 playing grid it can be thought of as a game that could be analyzed to a finish. Connect 4 is a variation on noughts and crosses limiting the player's possible moves to filled columns only and the game is made more difficult because a line of four is required. The classic two-player board game is chess, but programming a full chess implementation is beyond a one-chapter tutorial. The game Go is such an easy game to learn and such a difficult one to master. The game we are going to concentrate on is a simple game to learn and again quite tricky to play well, but it does have some very useful ground rules that makes it easier to teach a computer to play at least reasonably well.

The game of Reversi

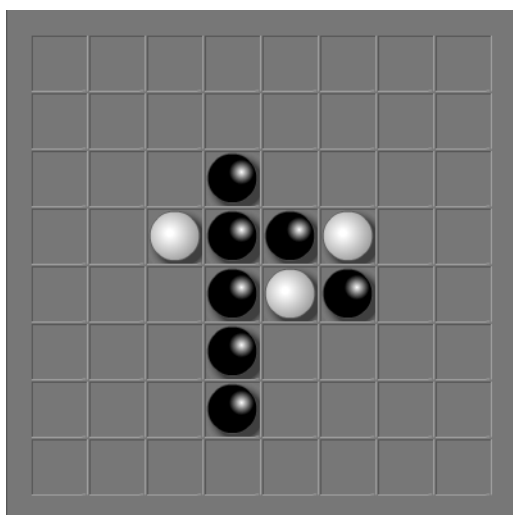


Figure 13.1 *The traditional game Reversi*

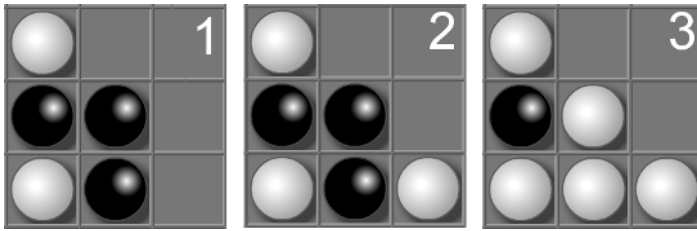


Figure 13.2 *Placing a new piece*

See Figure 13.1. The traditional game Reversi is sold today as the game ‘Othello’. It has very simple rules, yet the game has all the hallmarks of a game of skill, namely that a good player will always beat a poor player; luck does not play any role. Each player uses the same counters; these are colored differently on each side, black and white being two obvious choices. One player always places their counters down with one particular side facing up and the other player uses the opposite side. The board is a grid of 64 squares. The game starts with each player taking it in turns to fill the four central squares with playing pieces. After these first four pieces are placed, the player must sandwich their opponent’s pieces between their own colored pieces in a straight line. A line can be vertical or diagonal. Any opponent’s piece that is sandwiched between the player’s new piece and any existing player pieces are flipped over.

Figure 13.2 shows how placing a white piece at the lower right corner has the effect of flipping two black pieces. A line of opposing pieces can be of any length. The game is completed when either all the squares on the board are filled or a player cannot go. So that’s the game. Where do we begin to program this game to function as a solo player against the computer style activity?

Methods for board games

First we need to reduce the problem into manageable chunks. How are we going to play the game?

- Build and display the board.
- Check for user input.
- If the user places a piece legally, update the placed square and then update the board.
- Scan the board for legal computer moves.
- Work out the best move for the computer.
- Place a computer piece in the best-calculated square and then update the board.
- Repeat steps 2–6 until the game ends.
- Show the number of player pieces and computer pieces and declare a winner.

So we will need an initialization function to build a clear board, a legal move generator and an evaluation function to determine the best computer move. By far the hardest part of this is the evaluation function. Such a function could in fact look ahead to the end of the game and work out which move from the current position would give the computer the best result. We are only going to look ahead two moves; the idea for the two moves is to maximize the computer’s result and minimize the player’s. If you are interested in the technical background to these types of

games then try a search for ‘combinatorial game theory’ in your favorite web search engine; <http://www.ics.uci.edu/~epstein/cgt/> is a good starting point.

Building and initializing the board

Without question this is the easiest part of the programming. The game is a simple 8×8 grid; the images that can appear in this grid are limited to just three possibilities: a blank square, a white piece or a black piece. Figure 13.3 shows the images needed. To achieve this we will create three bitmaps that are duplicated to form the 8×8 grid. Each cell of the grid can be one of just three images: empty, containing a white piece or containing a black piece.

Listing 13.1 shows the simple code used in the ‘startMovie’ handler. Take a look at ‘Examples\Chapter13\reversi.dir’ to examine the code; it is cast member 1. In this Chapter we are going to use JavaScript for all the scripting.

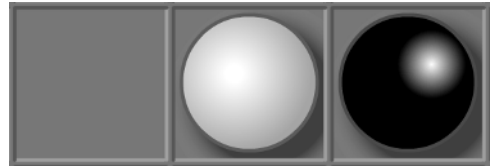


Figure 13.3 Three frames are used to display the pieces

```

1 var gPLAYER, gCOMPUTER, gFlip, gBoard, gCBoard, gPBoard, gCurGo;
2 var gWhoseGo;
3
4 function startMovie(){
5     gPLAYER = 3;
6     gCOMPUTER = 2;
7     gFlip = new Array(8);
8     gBoard = new Array(64);
9     gCBoard = new Array(64);
10    gPBoard = new Array(64);
11    initGame();
12 }
```

Listing 13.1

Arrays are used throughout the program to store the data needed. The array ‘gFlip’ stores which directions from a given square have a line that can be flipped after executing a legal move. ‘gBoard’ contains the current view of the board in a computer friendly manner. ‘gCBoard’ stores the temporary board after the computer has made a possible move and is used in the code for evaluating the computer’s best option. ‘gPBoard’ stores the player’s move following the computer’s move and is used for evaluating the player’s least-best move after the computer has made a temporary move. We will look at the use of these arrays in more detail later in the chapter. Then we have a call to a simple function that initializes the game.

```

1 function initGame(){
2     for (var i=0; i<64; i++){
```

```

3      gBoard[i] = gCBoard[i] = gPBoard[i] = 0;
4  }
5
6  for (var i = 0; i<8; i++){
7      gFlip[i] = false;
8  }
9
10 updateBoard();
11 gCurGo = 0;
12 playersMove();
13 }

```

Listing 13.2

The initialization function calls the ‘updateBoard’ function, which actually paints the current state of the board based on the state of the array ‘gBoard’. If the array contains zero, then a blank cell image will be painted at that square, a value of ‘gPLAYER’ will cause a cell with a white counter to be painted and finally if the array has the value ‘gCOMPUTER’ then a cell with a black counter will be painted. The updateBoard function is given in Listing 13.3. Notice that with this movie we have no sprite on the score at all. All the screen updates are handled entirely in code. We use the ‘copyPixels’ method for the image object. The stage itself has an image object, which is retrieved using ‘_movie.stage.image’. The copyPixels method takes up to four parameters. The last parameter allows you to use clever ink effects. For this movie we simply use the default copy so the last parameter can be omitted. The first parameter is the image we wish to paint, the second is a rectangle defining where it is to be painted in the coordinates of the calling image object, in this case the stage, and the third is the source rectangle of the source tile in this case. The shape of the source rectangle does not vary so we can simply create this once and store it in a variable called ‘srcRect’. At line 13 of Listing 13.3 we position the rectangle on the stage. For a given row and column the rectangle is defined as

- *Left* – the current column times 45 (the width of a cell) plus an offset that places the board just in from the left edge.
- *Top* – the current row times 45 (the height of a cell) plus an offset that places the board centrally vertically.
- *Right* – the value for ‘Left’ plus 45 (the width of a cell).
- *Bottom* – the value for ‘Top’ plus 45 (the height of a cell).

Lines 16–26 select the appropriate source image based on the current value in the array gBoard.

```

1  function updateBoard(){
2      var id, blank, white, black, srcRect, destRect, orgX, orgY;
3
4      blank = member("Blank").image;

```



```

5  white = member("White").image;
6  black = member("Black").image;
7  srcRect = rect(0,0,45,45);
8  orgX = 20;
9  orgY = 20;
10
11  for(var row=0; row<8; row++){
12    for(var col=0; col<8; col++){
13      destRect = rect(col*45 + orgX, row*45 + orgY,
14                    (col+1)*45 + orgX, (row+1)*45 + orgY);
15      id = row * 8 + col;
16      switch(gBoard[id]){
17        case 0:
18          (_movie.stage).image.copyPixels(blank, destRect, srcRect);
19          break;
20        case gPLAYER:
21          (_movie.stage).image.copyPixels(white, destRect, srcRect);
22          break;
23        case gCOMPUTER:
24          (_movie.stage).image.copyPixels(black, destRect, srcRect);
25          break;
26      }
27    }
28  }
29 }

```

Listing 13.3

The initialization function also makes use of one of two functions that swap the current player and update the move counter 'gCurGo'. The variable 'gWhoseGo' is used to check for the next player, the next player is visually displayed using the red ball highlight. The appropriate image is used in the 'playersMove' and 'computerMove' functions.

```

1  function playersMove(){
2    var destRect, srcRect, orgX, orgY;
3
4    srcRect = rect(0,0,125,35);
5    orgX = 400;
6    orgY = 25;
7    destRect = rect(orgX, orgY, orgX + 125, orgY + 35);
8    (_movie.stage).image.copyPixels(member("ComputerOff").image,
9                                destRect, srcRect);
10   destRect = rect(orgX, orgY + 40, orgX + 125, orgY + 40 + 35);

```

```

11  (_movie.stage).image.copyPixels(member("PlayerOn").image,
12                                     destRect, srcRect);
13  destRect = rect(410, 345, 410 + 125, 345 + 35);
14  (_movie.stage).image.copyPixels(member("NewGameOff").image,
15                                     destRect, srcRect);
16  gWhoseGo = gCOMPUTER;
17  gCurGo++;
18 }
19
20 function computersMove(){
21  var destRect, srcRect, orgX, orgY;
22
23  srcRect = rect(0,0,125,35);
24  orgX = 400;
25  orgY = 25;
26  destRect = rect(orgX, orgY, orgX + 125, orgY + 35);
27  (_movie.stage).image.copyPixels(member("ComputerOn").image,
28                                     destRect, srcRect);
29  destRect = rect(orgX, orgY + 40, orgX + 125, orgY + 40 + 35);
30  (_movie.stage).image.copyPixels(member("PlayerOff").image,
31                                     destRect, srcRect);
32  destRect = rect(410, 345, 410 + 125, 345 + 35);
33  (_movie.stage).image.copyPixels(member("NewGameOff").image,
34                                     destRect, srcRect);
35  gWhoseGo = gPLAYER;
36  gCurGo++;
37 }

```

Listing 13.4

Tracking the player's move

A player's move is legal if (a) the square they clicked is empty; (b) an opponent's piece is in an adjacent square (left, right, top, bottom or any diagonal); and (c) continuing past the opponent's piece in the same direction there can be any number of opponent's pieces but eventually there must be a player's piece. This checking is all done using the 'checkBoard' function. This function can be called by either the player or the computer; the caller is contained in the parameter 'piece'. The function calls a further function 'scanBoard' a total of eight times. For each call the direction that is being scanned is contained in the fifth and sixth parameters. From any position on the board we need to look, using the points of a compass and going in a clockwise direction, north, north-east, east, south-east, south, south-west, west and north-west. The first two parameters contain the starting square and the third either the player or the opponent. For each direction

the array 'gFlip' is set to either true or false and a variable 'legal' dictates whether this is an acceptable square.

```

1  function checkBoard(row, col, piece){
2      var legal = false, opponent;
3
4      if (piece==gPLAYER){
5          opponent = gCOMPUTER;
6      }else{
7          opponent = gPLAYER;
8      }
9
10     if (scanBoard(row, col, piece, opponent, 0, -1)){//North
11         legal = true;
12         gFlip[0] = true;
13     }else{
14         gFlip[0] = false;
15     }
16     if (scanBoard(row, col, piece, opponent, 1, -1)){//North East
17         legal = true;
18         gFlip[1] = true;
19     }else{
20         gFlip[1] = false;
21     }
22     if (scanBoard(row, col, piece, opponent, 1, 0)){//East
23         legal = true;
24         gFlip[2] = true;
25     }else{
26         gFlip[2] = false;
27     }
28     if (scanBoard(row, col, piece, opponent, 1, 1)){//South East
29         legal = true;
30         gFlip[3] = true;
31     }else{
32         gFlip[3] = false;
33     }
34     if (scanBoard(row, col, piece, opponent, 0, +1)){//South
35         legal = true;
36         gFlip[4] = true;
37     }else{
38         gFlip[4] = false;
39     }
40     if (scanBoard(row, col, piece, opponent, -1, 1)){//South West

```

```

41     legal = true;
42     gFlip[5] = true;
43 }else{
44     gFlip[5] = false;
45 }
46 if (scanBoard(row, col, piece, opponent, -1, 0)){//West
47     legal = true;
48     gFlip[6] = true;
49 }else{
50     gFlip[6] = false;
51 }
52 if (scanBoard(row, col, piece, opponent, -1, -1)){//North West
53     legal = true;
54     gFlip[7] = true;
55 }else{
56     gFlip[7] = false;
57 }
58
59 return legal;
60 }

```

Listing 13.5

The 'scanBoard' function is given in Listing 13.6. The first two parameters give the starting square, the third and fourth define which piece is the one being scanned for and which is the opponent and the fifth and sixth parameters are used in a repeat loop to check for the existence of an opponent's piece. By using every combination of an increment or decrement in 'x' and 'y' we can scan the array 'gBoard' in each compass direction. A value of 0 for the x increment and -1 for the y would scan in the north direction, while a value of 1 for both x and y would scan in the south-east direction. There must be at least one piece so if none is found the function returns false. If there is an opponent's piece then the values for 'incX' and 'incY' are added repeatedly until the condition fails. At this point we check that the end of the line contains a player's piece; if this is true then this direction contains a line that is suitable.

```

1  function scanBoard(row, col, piece, opponent, incX, incY){
2      var x, y, id;
3
4      x = col + incX;
5      y = row + incY;
6      id = y * 8 + x;
7
8      //Must be at least one
9      if (gBoard[id] != opponent) return false;

```

```

10
11     while(gBoard[id]==opponent){
12         x += incX;
13         y += incY;
14         id = y * 8 + x;
15     }
16
17     //Next piece must be a piece frame
18     if (gBoard[id] == piece) return true;
19
20     return false;
21 }

```

Listing 13.6

Once the player's chosen square has been checked and found to be legal we need to update the display. The function 'adjustBoard' does that. The start location and current player are passed to the function and then the values stored in the 'gFlip' array are used. Again we use eight calls to another function with the direction passed to each call.

```

1  function adjustBoard(row, col, piece){
2      var opponent;
3
4      if (piece==gPLAYER){
5          opponent = gCOMPUTER;
6      }else{
7          opponent = gPLAYER;
8      }
9
10     if (gFlip[0]) flipLine(row, col, piece, opponent, 1, 0);
11     if (gFlip[1]) flipLine(row, col, piece, opponent, 0, 1);
12     if (gFlip[2]) flipLine(row, col, piece, opponent, -1, 0);
13     if (gFlip[3]) flipLine(row, col, piece, opponent, 0, -1);
14     if (gFlip[4]) flipLine(row, col, piece, opponent, -1, -1);
15     if (gFlip[5]) flipLine(row, col, piece, opponent, 1, -1);
16     if (gFlip[6]) flipLine(row, col, piece, opponent, 1, 1);
17     if (gFlip[7]) flipLine(row, col, piece, opponent, -1, 1);
18     updateBoard();
19 }

```

Listing 13.7

The function 'gFlipLine' does the actual work of manipulating the 'gBoard' array.

```

1  function gFlipLine(row, col, piece, opponent, incX, incY){
2      var x, y, id;
3
4      x = col + incX;
5      y = row + incY;
6      id = y * 8 + x;
7
8      while(gBoard[id]==opponent){
9          gBoard[id] = piece;
10         x += incX;
11         y += incY;
12         id = y * 8 + x;
13     }
14 }

```

Listing 13.8

The function ‘updateBoard’ is again used to repaint the display.

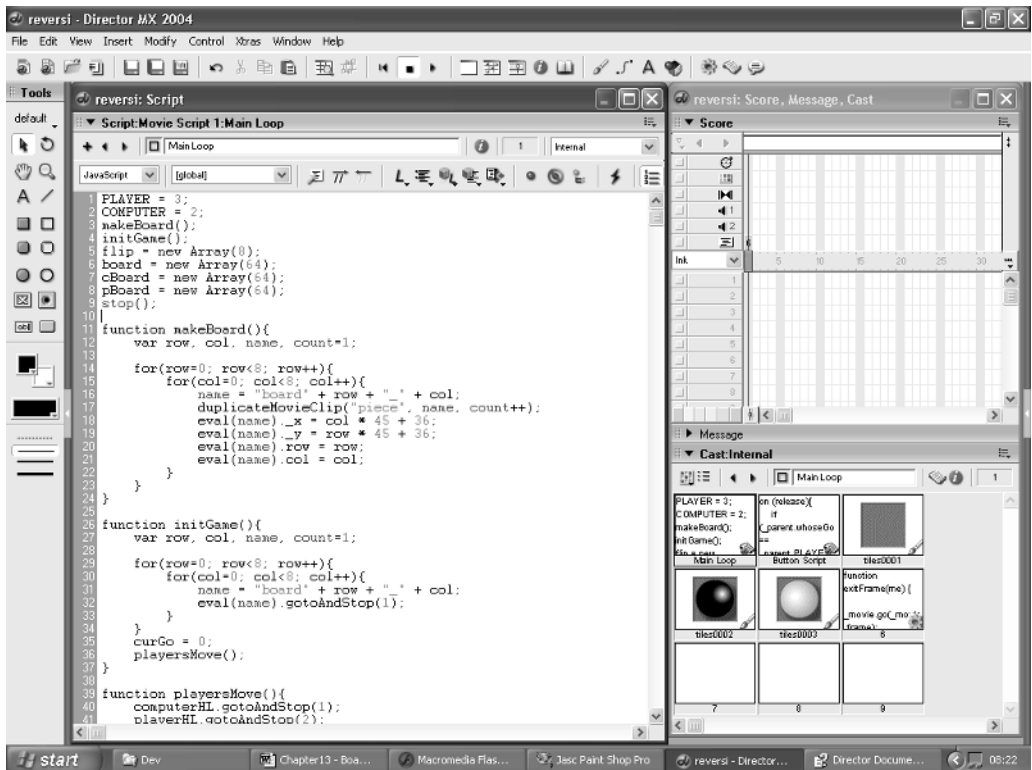


Figure 13.4 Developing the Reversi game

Evaluating the computer's best move

The evaluation function for the computer's best move is the most complex part of the game to code efficiently. The aim of the function is to derive the computer's best move based on a look ahead of two moves. We use the array 'gBoard' as the basis for all analysis. This array starts in the top left corner and scans the board one row at a time; index 8 is therefore row 1, column 0.

After storing the board position in a format that is easily readable by the computer we then process every square of the board looking for a legal move. The function used is 'computerScore'; this function returns -1 if the position is an illegal move and a score for the computer if the square is a legal move. We will look a little later at how this score is derived. If the move is legal then we use the position following this move, which is stored in the single-dimension array 'cBoard' in a further function 'playerScore'; the purpose of this function is to calculate the player's best score from the board position stored in the cBoard. Then we subtract the player's best score from the computer's best score and work out the move that will give the computer the best advantage. Finally, having discovered the best move on the basis of this simple algorithm, we calculate the row and column that the move implies. Because the arrays are single dimensional, to map to row and column we use the code snippet

```
row = Math.floor(id/8);
col = id % 8;
```

Here, 'row' is the integer value after dividing the index 'id' by 8, which will place row between 0 and 7. The value of 'col' is determined using the modulus operator '%'. This operator returns the remainder after the left of the operator is divided by the right. Again this will return a value between 0 and 7. For example, $19 \% 8 = 3$, because $19 \div 8 = 2$ with remainder 3 and the modulus returns this remainder. We update the 'gFlip' array by calling 'legalMove' again and set the current square directly; it then remains to update the on screen display using the 'adjustBoard' function we looked at earlier.

```
1 function doComputerMove(){
2     var diff=-100000, c, p, i, best=-1, id, row, col;
3
4     for (i=0; i<64; i++){
5         c = computerScore(i);
6         if (c > -1){
7             p = playerScore();
8             if (diff < (c - p) || best== -1){
9                 diff = c - p;
10                best = i;
11            }
12        }
13    }
14
15    row = Math.floor(best/8);
```

```

16     col = best % 8;
17     id = best;
18     legalMove(best, gCOMPUTER);
19
20     gBoard[id] = gCOMPUTER;
21
22     adjustBoard(row, col, gCOMPUTER);
23
24     playersMove();
25 }

```

Listing 13.9

The function in Listing 13.9 calls several functions, which we will look at in detail. The first function to consider is the ‘computerScore’ function. This function uses the ‘legalMove’ generator to test whether the current square is legitimate for the current board and player. If it is, then we set the array ‘gCBoard’ to be a duplicate of ‘gBoard’, which you will recall is simply the current displayed board. Then we update the current board using the function ‘adjustArray’. Finally, we count how many ‘gCOMPUTER’ pieces are on the new board, stored in the array gCBoard.

```

1  function computerScore(i) {
2      var n, count;
3
4      if (!legalMove(i, COMPUTER)) return -1;
5
6      for (n=0; n<64; n++) gCBoard[n] = gBoard[n];
7      gCBoard[i] = gCOMPUTER;
8      adjustArray(i, gCOMPUTER);
9
10     count = 0;
11     for (n=0; n<64; n++) {
12         if (gCBoard[n]==gCOMPUTER) count++;
13     }
14     return count;
15 }

```

Listing 13.10

The ‘legalMove’ function works in much the same way as the function ‘checkBoard’ only this function operates on the temporary arrays ‘gCBoard’ and ‘gPBoard’ rather than using the ‘gBoard’ array. The function in turn calls the ‘scanArray’ function, which is a mirror of ‘scanBoard’ operating on these temporary arrays.

```

1  function legalMove(i, piece) {
2      var legal = false;

```



```
3
4     if (scanArray(i, piece, 1, 0)){
5         legal = true;
6         gFlip[0] = true;
7     }else{
8         gFlip[0] = false;
9     }
10    if (scanArray(i, piece, 0, 1)){
11        legal = true;
12        gFlip[1] = true;
13    }else{
14        gFlip[1] = false;
15    }
16    if (scanArray(i, piece, -1, 0)){
17        legal = true;
18        gFlip[2] = true;
19    }else{
20        gFlip[2] = false;
21    }
22    if (scanArray(i, piece, 0, -1)){
23        legal = true;
24        gFlip[3] = true;
25    }else{
26        gFlip[3] = false;
27    }
28    if (scanArray(i, piece, -1, -1)){
29        legal = true;
30        gFlip[4] = true;
31    }else{
32        gFlip[4] = false;
33    }
34    if (scanArray(i, piece, 1, -1)){
35        legal = true;
36        gFlip[5] = true;
37    }else{
38        gFlip[5] = false;
39    }
40    if (scanArray(i, piece, 1, 1)){
41        legal = true;
42        gFlip[6] = true;
43    }else{
44        gFlip[6] = false;
45    }
```

```

46     if (scanArray(i, piece, -1, 1)){
47         legal = true;
48         gFlip[7] = true;
49     }else{
50         gFlip[7] = false;
51     }
52
53     return legal;
54 }

```

Listing 13.11

The function can operate either on the ‘board’ array if the value for ‘piece’ is ‘COMPUTER’ or the ‘cBoard’ array if the value for ‘piece’ is ‘PLAYER’. Again when scanning a row the first square must be empty, the next square must contain an opponent piece and the final square must be a player piece. If all the conditions are passed then true is returned, otherwise false is returned.

```

1  function scanArray(i, piece, incX, incY){
2      var name, n, inc;
3
4      inc = incY*8 + incX;
5      n = i + inc;
6
7      //Must be at least one
8      switch (piece){
9          case gPLAYER:
10             if (gCBoard[i]!=1 || gCBoard[n]!=gCOMPUTER) return false;
11             do{
12                 n += inc;
13             }while(gCBoard[n] == gCOMPUTER);
14             //Next piece must be a PLAYER piece
15             if (gCBoard[n] == gPLAYER) return true;
16             break;
17
18             case gCOMPUTER:
19                 if (gBoard[i]!=1 || gBoard[n]!=gPLAYER) return false;
20                 do{
21                     n += inc;
22                 }while(gBoard[n] == gPLAYER);
23                 //Next piece must be a COMPUTER piece
24                 if (gBoard[n] == gCOMPUTER) return true;
25                 break;
26             }
27

```

```

28     return false;
29 }

```

Listing 13.12

Once the best computer score is evaluated the function 'adjustArray' is called.

```

1  function adjustArray(i, piece){
2      if (gFlip[0]) flipArrayLine(i, piece, 1, 0);
3      if (gFlip[1]) flipArrayLine(i, piece, 0, 1);
4      if (gFlip[2]) flipArrayLine(i, piece, -1, 0);
5      if (gFlip[3]) flipArrayLine(i, piece, 0, -1);
6      if (gFlip[4]) flipArrayLine(i, piece, -1, -1);
7      if (gFlip[5]) flipArrayLine(i, piece, 1, -1);
8      if (gFlip[6]) flipArrayLine(i, piece, 1, 1);
9      if (gFlip[7]) flipArrayLine(i, piece, -1, 1);
10 }

```

Listing 13.13

This uses the function 'gFlipArrayLine'.

```

1  function flipArrayLine(i, piece, incX, incY){
2      var n, inc;
3
4      inc = 8*incY + incX;
5      n = i + inc;
6
7      switch (piece){
8          case gPLAYER:
9              while(pBoard[n]==gCOMPUTER){
10                 gPBoard[n] = gPLAYER;
11                 n += inc;
12             }
13             break;
14
15             case gCOMPUTER:
16                 while(gCBoard[n]==gPLAYER){
17                     gCBoard[n] = gCOMPUTER;
18                     n += inc;
19                 }
20                 break;
21             }
22 }

```

Listing 13.14

This leaves just the 'playerScore' function to consider from the main function 'doComputerMove'. This works in much the same way as calculating the computer's best move. Each square in the board stored in the array 'cBoard' is tested for a legal 'PLAYER' move. Subject to this test evaluating to true we update the 'pBoard' array and count the number of PLAYER squares; if this is a higher value than the stored value for 'score' then score is updated. After considering every square the final value for score is returned.

```

1  function playerScore(){
2      var i, n, score=0, count;
3
4      for (i=0; i<64; i++){
5          if (legalMove(i, gPLAYER)){
6              for(n=0; n<64; n++) gPBoard[n] = gCBoard[n];
7              adjustArray(i, gPLAYER);
8              gPBoard[i] = gPLAYER;
9              count = 0;
10             for(n=0; n<64; n++){
11                 if (gPBoard[n]==gPLAYER) count++;
12             }
13             if (count>score) score = count;
14         }
15     }
16
17     return score;
18 }
```

Listing 13.15

After running the code we are left with a combination of the computer's best move that will lead to the player's worst move. You could amend the functions to calculate this to a greater depth. You could look ahead several moves and calculate the computer move that will lead to the best combination of the computer's best move and the player's worst move.

Using this code gives a reasonable game. Problems do occur however, stemming from the evaluation function using array values out of range. This is an important point. Whenever you use arrays in this way, it is always best to ensure that the bounds are within the constraints of the array. A call to an array with an index below zero or above the length of the array - 1 returns an undefined value. It is easy to mistake this for useful data and so it is always best to check that the index for an array is within the bounds of the array.

Improving the evaluation function

The score derived in this function is all about the pieces on the board after two look-ahead moves. Although this gives some semblance of intelligence, a good Reversi player knows that certain rules improve their play. In the early stages of the game, controlling the four central squares is a useful

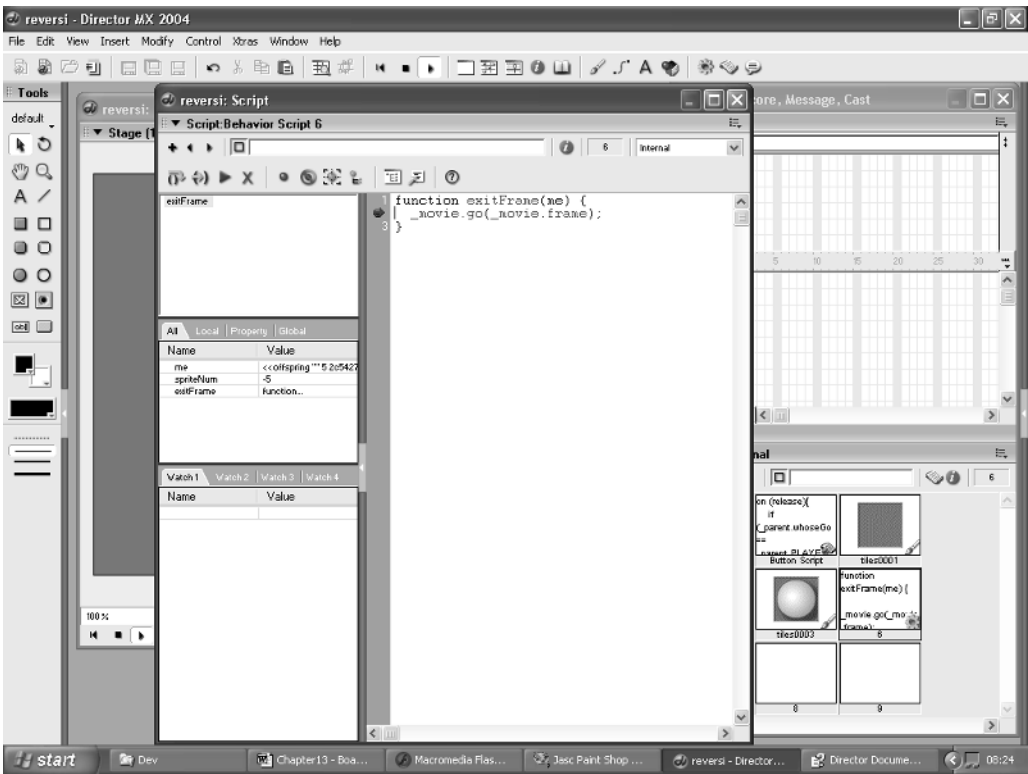


Figure 13.5 *Debugging the game*

extra, so you could add to your evaluation function by scoring these squares higher in the return value for a position. Also a border square means that your opponent cannot get a piece to the other side of you, so they are more valuable; again using this positional information you could add to the score in these circumstances. The three squares that surround a corner are very poor squares to play in because your opponent will almost certainly then get a corner and corner pieces can never be 'gFlipped'. You could improve the evaluation function greatly by scoring these squares very low, regardless of the short-term gain they may achieve. Finally, getting a corner is the strongest move you can play and should be given a very high score. These improvements in the evaluation function will make a huge difference to the chances of a poor player ever beating the computer. When creating this type of game the method is always to look for ways to improve the evaluation function while not performing an exhaustive search. In many games an exhaustive search is well beyond the power of today's computers. The number of possible game options from a single-game position is enormous and working through the game to the end for each of these is not a realistic possibility.

The presentation

Although the presentation of the game in this chapter was quite traditional, it doesn't have to be that way. The game of Reversi could be presented using a cat and mouse theme; you place a cat and

all the mice in a line run-off and a cat takes their place. A cartoon effect on the changeover would make the game strikingly different while the game play remains the same.

Summary

In this chapter you were given an overview of the techniques required to create a two-player board game where the computer is a reasonable opponent. The techniques we learnt were board initialization, legal move generation and evaluation functions. The final technique is where your skills as a programmer and all your creativity will be used. After the mind games in this chapter you will find the manual dexterity of the next a welcome break.

14 Quizzes

Quizzes are a very popular form of entertainment on TV. In the UK, my home country, there is a quiz show called ‘Who wants to be a millionaire?’ which rapidly became one of the highest-rated programmes on TV and was successfully sold all around the world. Quizzes are very simple to create, but they are fairly demanding to maintain. In this chapter we will look at creating a database of questions using an Access database. The topics covered in this chapter involve using server-side scripting. If you intend to create games to be delivered on the web then many of the ideas in this chapter will be suitable for storing and maintaining high-score tables and other persistent server-side data. Databases are a very flexible way of storing data and well worth learning the about. You will almost certainly find that the simplest databases will suit your game requirements. In the quiz example that forms the basis of this chapter, the questions are categorized by difficulty and category, allowing us to use the database to access questions in a variety of ways.

Multiple choice or free text?

The easiest way to write a quiz program is to use questions that have multiple-choice answers. The advantage of using multiple choice is that any answer the user gives is either *definitely* wrong or *definitely* right. There is no ambiguity. Computer programs hate ambiguity. The disadvantage is that the quiz compilers are required to create the alternative ‘wrong’ answers, which adds to their work. Another option is free-text input, but this is prone to errors. A quiz that allows free-text input for the answers must have answers that are very easy to input. Numeric answers are the least prone to error, but even here there are possible problems; if the answer is ‘3’, how do you respond to an answer of ‘three’? Both answers are correct but if the code is checking for 3 and finds three then the player could easily be marked wrong. Free-text input is very demanding to code; many answers that are incorrectly spelt must be marked as correct. But how do you allow for that? One way is to compare the player’s answer with the correct answer using a table of permissible, wrongly spelled answers. But however you approach it, the problem is complex and can result in ‘obvious’ wrong answers being marked correct and vice versa. In this chapter we are, therefore, only going to consider the multiple-choice option. In the database of quiz questions we will keep the correct answer and three incorrect ones for each question.

Creating a database

Databases come in many shapes and forms. In this chapter we will concentrate on using Access. This is a relational database management system for Windows. The principal alternative on Windows is SQL Server. If you intend to do much database work then using SQL Server is recommended. You would need to get a copy of the developer edition. Unfortunately this costs

about the same as a Director upgrade although a trial evaluation version is available. A simpler alternative is to use an Access-based database. Access comes with a complete version of Microsoft Office and much of the material in this chapter applies to both databases. The major difference between the two options is in the way the server implements the database access. The SQL Server route is much more flexible and results in a smoother experience for the user. In the end the cost difference is more than outweighed by the benefits.

Because most of this chapter involves the use of server-based techniques the examples do not run direct from the CD. They are on the CD so that you can examine the code, but to use them in a live way you will need to be running Internet Information Services (IIS). If you are using a Windows platform it is easy to set up your own machine to include IIS. Check out the documentation that comes with your version of Windows to learn how to do this (unfortunately the Home Edition of Windows XP does not allow you to set up your machine to use IIS). If you are unable to set up your machine for IIS either because your version of Windows is unsuitable or because you are developing on a Mac then you will need to upload the ASP (Active Server Pages) stuff to an Internet service provider and run the samples across the Internet.

Step 1

First we will create a database. For the examples in this chapter I have created a database called 'Quiz'. Simply click on the 'New/Blank Database' button to the right of the default screen layout or you can open the database from the CD by choosing 'Open a file/More Files...' and navigating to 'Examples/Chapter14/quiz.mdb' (see Figure 14.1).

Step 2

The next step is to add a table to the database. A database can consist of many tables and you can create new tables that are simply a combination of rules to join other tables together. In this very simple example we need a single table that we will call 'Questions'. To create a table you can use several approaches; we are going to use 'Design View'. Click the 'Create table in Design view' link (see Figure 14.2).

Step 3

We now have a database with a single table. In the table we need to add fields (see Figure 14.3). These are where the actual data will be stored. For this example we add the following.

- *Difficulty* – a number that uses a single byte (0–255 range).
- *Category* – a string value that can contain up to 50 characters.
- *Question* – a string value that can contain up to 255 characters.
- *A, B, C, D* – answers, all string values that can contain up to 80 characters. A is the correct answer.

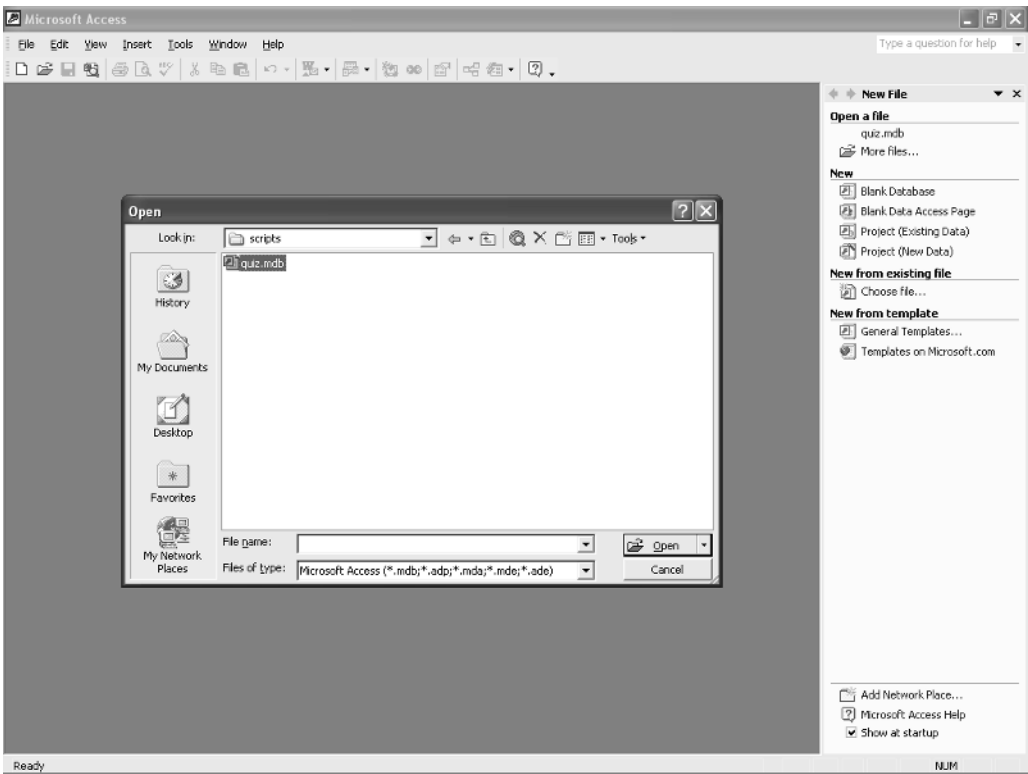


Figure 14.1 Creating a database with Microsoft Access, Step 1

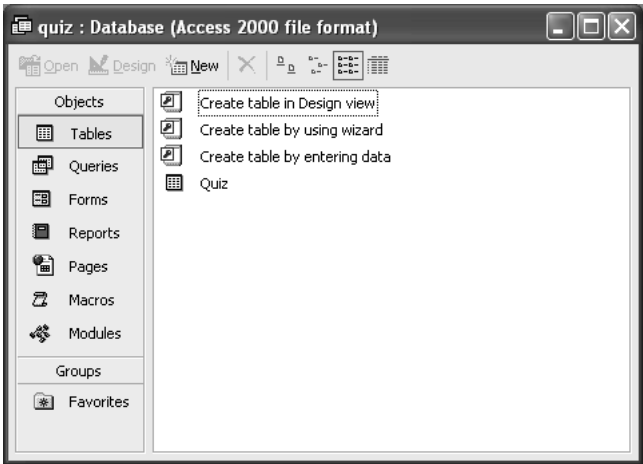


Figure 14.2 Creating the database, Step 2 – creating a table using ‘Design View’

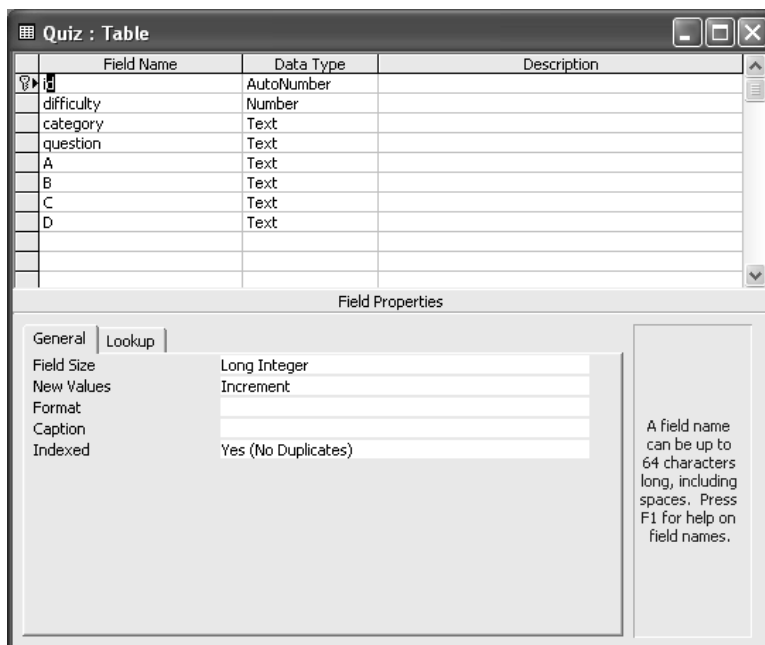


Figure 14.3 *Creating the database, Step 3 – adding fields to the table*

There are many options for the datatypes and it is at the design stage that you need to decide what the most suitable datatypes will be for each field in all your tables. The larger you set the datatypes the faster the database will grow in file size.

Step 4

Having set up the table, click 'close' and choose to save the table. You will see the table listed in Access's main window. Click on it and you will get a window like in Figure 14.4. You can add questions directly into this window. But the purpose of this chapter is to show how we can use Director to build the database and then present views of the database to on-line players.

See how easy it is to set up a database.

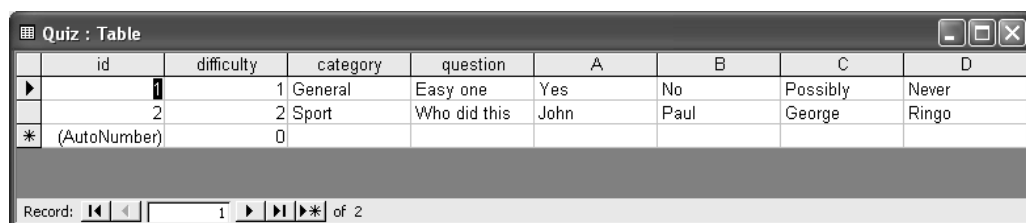


Figure 14.4 *Creating the database, Step 4*

Inputting the data

To make the inputting of data easy we will create a Director-based input tool. Open 'Examples\Chapter14\createquiz.dir'. This project uses the new Flash UI components in Director to create a tool that allows the developer to read and update the database. Access to the database is via ASP pages. ASP pages can run on your local machine if it is set up to use IIS or you can run them on a remote server. Before we look at how the createquiz.dir project works we will look at the ASP pages it uses.

Connecting to the database using ASP

When using a database the first thing we must do is to connect to it. Because all our pages will use it, we will put this code in a file that can be included in all the other files. To create a connection we can use JavaScript. If you have followed the JavaScript samples that have appeared throughout the book then the syntax should be familiar. ASP pages start with the '<% ' tag indicating a block of code and a code block will finish with a '%>' tag. First we declare two variables at line 7, 'connStr' and 'conn'. We can use the 'Server' object to create an 'ADODB.Connection', using the 'CreateObject' method. We can create a connection string, which is simply a string containing all the information a database connection will require. The string shown in lines 11 and 12 is suitable for a simple Access database. Although by this stage we have a connection object, it is not yet connected; to actually connect we must use the 'Open' method of the connection object passing the connection string as a parameter. Here is the code, which can be found as 'Examples\Chapter14\ scripts\dbconnect.asp'.

```

1  <%
2  //-----
3  //Connects to database
4  //-----
5
6  //*****DB connection*****
7  var connStr, conn;
8
9  //Connect to the database
10 conn = Server.CreateObject("ADODB.Connection");
11 connStr = "PROVIDER=Microsoft.Jet.OLEDB.4.0; DATA SOURCE=" +
12           Server.MapPath("quiz.mdb") + ";" + "Password=";
13 conn.Open(connStr);
14
15 %>
```

Listing 14.1

Creating a new row

Now that we have the script to create a connection, we can use it to create a new entry in the database or to update an existing entry. The details for the new entry are passed in via a

querystring. A querystring takes the form

```
varName1=varValue1&varName2=varValue2&varName3=varValue3&
varName4=varValue4&...
```

Each variable name and value pair is linked using the ampersand character. To turn the string into variable data you can use the 'Request' command. The first section of the script turns the querystring into the variables 'category', 'difficulty', 'question' and 'A, B, C, D'.

An ASP page is cached by the browser so it is best to include the command 'Response.Expires = 0;' at the beginning of the file to ensure that this caching does not take place, otherwise the page can only be used once.

Having parsed the string into usable variables the 'dbconnect.asp' file is included in the file. The effect of an include is as though the code in the file is actually part of this page. So if everything has worked at this stage in the file you will have the variables to define a record in the database and an open connection. Because the same ASP page is used for both updating and adding we must first find out whether this question appears in the database. To do this we will need another object, an 'ADODB.recordset', which is created in the same way as the connection. Having created a recordset we use the 'Open' method passing a string and the open connection. This time the string is an SQL query. To discover whether the record exists we simply ask for a list of records containing this question using the SQL query 'SELECT * FROM quiz WHERE question = [the current question]'. Here the table is 'quiz' and the only test is for the single parameter 'question'. The database engine then looks through all the records in the quiz table for a match for the current question. Each time it finds a match it is added as a new record to the object 'rs'. We can move between records in the rs object until we reach the end when 'rs.EOF' (end of file) evaluates to true. If rs.EOF is false after the SQL query then the question was not found. To allow us to use this information later in the code the variable 'found' is set to true or false based on the return value of the query: true if the question exists and false if not.

We are nearly ready to update or add to the database. First we must format a query string using the information we have discovered. If we are updating then we use the SQL command 'UPDATE'. The format is to follow UPDATE with the table name and then use 'SET' and a list of values to set separated by commas. So that SQL knows which record to update the query is completed by adding 'WHERE' with the column name 'question' being equal to the current variable 'question'. If instead we are adding a new record then we use the SQL command 'INSERT INTO' followed by the table name and a list of column names separated by commas inside round brackets. The value of the column names follows the word 'VALUES' and is a list of values separated by commas inside round brackets in the same way as the column names. To actually send this information to the database, we create the string then pass it to the 'Execute' method of the open connection. Finally we close the open connection and pass some data to Director, either 'questionAdded' or 'questionUpdated' using the 'Response. Write' method. Inside Director the text will be received using a 'netTextResult' call. More on this will be covered later. The code for this script is in the file 'Examples\Chapter14\scripts\addQuestion.asp'.

```
1 <%@Language=JavaScript %>
```

```
2
```

```

3  <%
4  //-----
5  //Script writes a new question to the database from a querystring
6  //-----
7  //Strip data from the query string
8  Response.Expires = 0;
9  var category = Request("category");
10 var difficulty = Request("difficulty");
11 var question = Request("question");
12 var A = Request("A");
13 var B = Request("B");
14 var C = Request("C");
15 var D = Request("D");
16
17 //-----DATABASE STUFF-----
18 %>
19 <!--#include file="dbconnect.asp"-->
20 <%
21 var rs=Server.CreateObject("ADODB.recordset");
22 var found = false;
23
24 rs.Open("SELECT * FROM quiz WHERE question=\'" + question + "
        \'", conn);
25 if (!rs.EOF) found = true;
26 rs.Close();
27
28 var sqlStr, date;
29 var d = new Date();
30 var dateStr = new String((d.getMonth()+1) + "/" + d.getDate() +
31                          "/" + d.getYear());
32
33 if (found) {
34     // The entry exists so we will amend it
35     sqlStr = "UPDATE quiz SET category=\'" + category +
36             "\', difficulty=" + difficulty + ", A=\'" + A +
37             "\', B=\'" + B + "\', C=\'" + C + "\', D=\'" + D +
38             "\' WHERE question=\'" + question + "\'";
39     Response.Write("questionUpdated");
40 }else{
41     //No entry so add it
42     sqlStr = "INSERT INTO quiz " +
43             "(category, difficulty, question, A, B, C, D, create
        date) " +

```

```

44         "VALUES (\'" + category + "\", " + difficulty + ", \'" +
45         question + "\", \'" + A + "\", \'" + B + "\", \'" + C +
46         "\", \'" + D + "\", \'" + dateStr + "\')";
47     Response.Write("questionAdded");
48 }
49
50 conn.Execute(sqlStr);
51 conn.Close();
52
53
54 %>

```

Listing 14.2

Deleting a row

Sometimes we will need to delete a row. Here the method is similar to above. First we parse the querystring to find the current question. Then we make a connection by including the 'dbconnect.asp' file. Then we build a query, this time using 'DELETE FROM' and the table name 'WHERE question = [the current value of the variable question]'. We can execute this directly on the database without the need for any recordsets. The code for this is in the file 'Examples\Chapter14\scripts\deleteQuestion.asp'.

```

1  <%@Language=JavaScript %>
2
3  <%
4  //-----
5  //Script writes a new question to the database from a querystring
6  //-----
7  //Strip data from the query string
8  Response.Expires = 0;
9  var question = Request("question");
10
11 //-----DATABASE STUFF-----
12 %>
13 <!--#include file="dbconnect.asp"-->
14 <%
15 var sqlStr = "DELETE FROM quiz WHERE question=\'" + Request.QueryString("question") +
16         question + "\'";
17 conn.Execute(sqlStr);
18 conn.Close();
19

```

```
20 Response.Write("questionDeleted = true");
21 %>
```

Listing 14.3

Deleting an entire category

For convenience another ASP page deletes an entire category. This is done in the same way as deleting a single question, except multiple records can be deleted. The code for this is in the file 'Examples\Chapter14\scripts\deleteCategory.asp'.

```
1  <%@Language=JavaScript %>
2
3  <%
4  //-----
5  //Script writes a new question to the database from a querystring
6  //-----
7  //Strip data from the query string
8  Response.Expires = 0;
9  var category = Request("category");
10
11 //-----DATABASE STUFF-----
12 %>
13 <!--#include file="dbconnect.asp"-->
14 <%
15 var sqlStr = "DELETE FROM quiz WHERE category=\'" + category + "\'";
16
17 conn.Execute(sqlStr);
18 conn.Close();
19
20 Response.Write("questionDeleted = true");
21 %>
```

Listing 14.4

Accessing a category

The input engine will require data from the records in a category. So that the user can access an entire category, the script 'Examples\Chapter14\scripts\getCategory.asp' was included. Here a connection is made to the database in the usual way and the variable 'category' is tripped from the querystring. The recordset is made using the SQL query 'SELECT * FROM' the table 'Quiz' 'WHERE category = [the value of the variable category]'. Then we create a string from the records to pass back to Director. The value of the columns in a record can be accessed using 'Fields.Item(x)' where 'x' is the number of the column. To set up the text suitable for Director we put each question in the following format:

- question
- difficulty
- correct
- option1
- option2
- option3.

Each question takes up six lines. The first line of the text is set to the number of questions in the category; this is set at line 36 of Listing 14.5. To move through the records in the recordset object use the 'moveNext' method as shown at line 33.

```

1  <%@ Language=JavaScript%>
2
3  <%
4  //-----
5  //Script writes card data to db and sends ecards
6  //-----
7  Response.Expires = 0;
8  var category;
9
10 //Strip data from the query string
11 category = Request("category");
12
13 //-----DATABASE STUFF-----
14 %>
15 <!--#include file="dbconnect.asp"-->
16 <%
17 var rs = Server.CreateObject("ADODB.Recordset");
18 var str = "SELECT * FROM Quiz WHERE category =\'" + category + "\'";
19
20 rs.Open( str, conn);
21
22 var i=0;
23 str = "";
24
25 while (!rs.EOF){
26     //Set string
27     str += (rs.Fields.Item(1).Value + "\n");
28     str += (rs.Fields.Item(3).Value + "\n");
29     str += (rs.Fields.Item(4).Value + "\n");
30     str += (rs.Fields.Item(5).Value + "\n");
31     str += (rs.Fields.Item(6).Value + "\n");
32     str += (rs.Fields.Item(7).Value + "\n");

```



```

33     rs.moveNext();
34     i++;
35 }
36 str = str + i;
37 Response.Write(str);
38
39 rs.Close();
40 conn.Close();
41
42 %>

```

Listing 14.5

Getting a list of all categories

The last script we will look at returns all the categories in the database, useful for creating the input engine. Much of the methodology will be familiar. The new feature is the use of the term ‘DISTINCT’, which ensures that only one record for each instance is returned; if there are 25 records in the ‘sport’ category only a single record sport is returned to the recordset. The data is passed back to Director using a single line for each category entry.

```

1  <%@Language=JavaScript %>
2
3  <%
4  //-----
5  //Script gets all the distinct category names from the database
6  //-----
7  //-----DATABASE STUFF-----
8  %>
9  <!--#include file="dbconnect.asp"-->
10 <%
11 Response.Expires = 0;
12 var rs = Server.CreateObject("ADODB.Recordset");
13 var str = "SELECT DISTINCT(category) FROM Quiz";
14
15 rs.Open( str, conn);
16
17 var i=0;
18 str = "";
19
while(!rs.EOF){
    str += ("category" + i + "=" + rs.Fields.Item(0).Value + "&");
    rs.moveNext();
    i++;

```

```

}
str += ("categoryTotal=" + i + "&loaded=1&");

rs.Close();
conn.Close();

Response.Write(str);

%>

```

Listing 14.6

Creating the front end

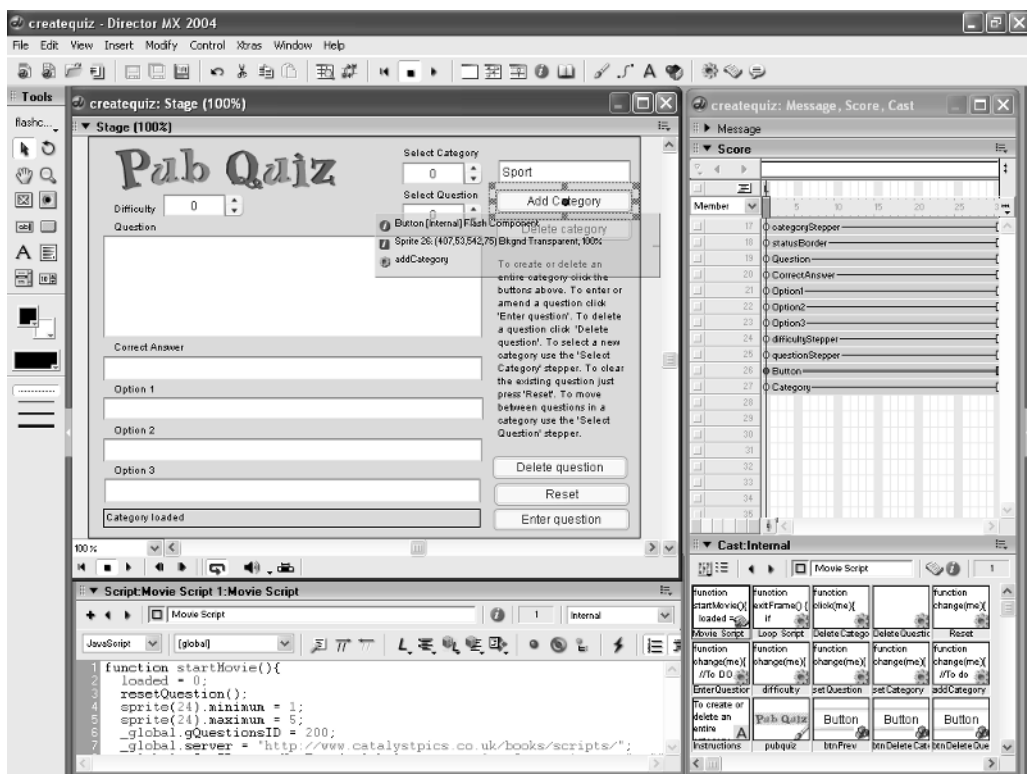


Figure 14.5 Creating an input application

Now we have the ASP scripts we can use them with a front end for inputting the data. In this example we will make use of the Flash UI components that come with Director MX 2004. The input screen has a numeric stepper control in the top middle to select the category and a numeric stepper just below this to move through the questions within a category. There are five buttons

on the right, 'Add category', 'Delete category', 'Delete question', 'Reset' and 'Enter question'. There is a numeric stepper to select the level of difficulty and finally six text boxes to enter and display the category, the question, the correct answer and three alternative answers. To initialize the data we call 'scripts/getCategories.asp' in the 'startMovie' function. This example uses JavaScript syntax throughout. Listing 14.7 shows the movie script for this movie. We will look initially at the startMovie function (lines 1–10 of Listing 14.7). First we set a global flag to indicate that we are not ready to display data. Then we clear the existing text in the question and answer text input filed using the 'resetQuestion' handler, which can be seen between lines 17 and 24. The text input files are all Flash UI components, you can get at the text in them by referring to the sprite channel and setting the 'text' property. Sprite 24 is the difficulty stepper. A stepper takes a minimum and maximum value then allows the user to move easily between these values by clicking the up and down arrows. The current value of the stepper is stored in a property variable called 'value'. At line 6 we set the value of the global variable 'gQuestionsID' to 200 and set the server to point to somewhere where the scripts are stored on a server that is capable of delivering ASP. Line 8 is where we try to get the text from the ASP page. We assign the value returned from this to the global variable 'gCatID'. Finally at line 9 we set the status text, which appears at the base of the screen. We will look at the remaining functions as we use them.

```

1  function startMovie(){
2    _global.init = false;
3    resetQuestion();
4    sprite(24).minimum = 1;
5    sprite(24).maximum = 5;
6    _global.gQuestionsID = 200;
7    _global.server = "http://www.catalystpics.co.uk/books/scripts/";
8    _global.gCatID = getNetText(_global.server + "getCategories.asp", "");
9    member("status").text = "Loading category list";
10 }
11
12 function stopMovie(){
13   resetQuestion();
14
15 }
16
17 function resetQuestion(){
18   sprite(24).value = 1;
19   sprite(19).text = "";
20   sprite(20).text = "";
21   sprite(21).text = "";
22   sprite(22).text = "";
23   sprite(23).text = "";
24 }
25

```

```

26 function addQuestion(c, i){
27   question = new Object();
28   question.question = "";
29   question.difficulty = 0;
30   question.correct = "";
31   question.option1 = "";
32   question.option2 = "";
33   question.option3 = "";
34   var str = "";
35   i++;
36   while (1){
37     if (c.charCodeAt(i)==13) break;
38     str += c.charAt(i);
39     i++;
40   }
41   i++;
42   question.difficulty = Number(str);
43   while (1){
44     if (c.charCodeAt(i)==13) break;
45     question.question += c.charAt(i);
46     i++;
47   }
48   i++;
49   while (1){
50     if (c.charCodeAt(i)==13) break;
51     question.correct += c.charAt(i);
52     i++;
53   }
54   i++;
55   while (1){
56     if (c.charCodeAt(i)==13) break;
57     question.option1 += c.charAt(i);
58     i++;
59   }
60   i++;
61   while (1){
62     if (c.charCodeAt(i)==13) break;
63     question.option2 += c.charAt(i);
64     i++;
65   }
66   i++;
67   trace(i + ", " + c.length);
68   while (1){

```

```

69     if (c.charCodeAt(i)==13||i>=c.length) break;
70     question.option3 += c.charAt(i);
71     i++;
72 }
73 _global.questions.push(question);
74 return i;
75 }
76
77 function setQuestion(id){
78     sprite(24).value = _global.questions[id-1].difficulty;
79     sprite(19).text = _global.questions[id-1].question;
80     sprite(20).text = _global.questions[id-1].correct;
81     sprite(21).text = _global.questions[id-1].option1;
82     sprite(22).text = _global.questions[id-1].option2;
83     sprite(23).text = _global.questions[id-1].option3;
84 }
85
86 function setCategory(id){
87     id--;
88     _global.gQuestionsID = getNetText(_global.server +
89         "getCategory.asp?category=" +
90         _global.categories[id], "");
91     member("Category").text = "Loading";
92     member("status").text = "Loading questions list";
93 }

```

Listing 14.7

The main loop

Having started the movie and called the 'getNetText' method we now go into a loop defined by the frame script on frame 1. Listing 14.8 shows the script. Line 2 shows how we retrieve the text from the 'startMovie' call to 'getCategories.asp'. When we used getNetText we set a global variable '_global.gCatID'. The method 'netDone' evaluates to true when the text with this ID has been downloaded. It is possible that the download is complete but that errors occurred so we also check for the 'netError' value for this ID being 'OK'. We are now assured that data is available and that no errors took place in the transfer. We now need to parse this data into a format that is suitable for our UI components. First, at line 3 we save the text into a variable 'c'. The 'netTextResult' method takes an ID parameter and returns the text associated with that ID. At line 5 we create a global variable called 'categories' and set it to be a new array. Line 6 initializes a temporary string variable 'str' to an empty string. We then enter a 'for' loop between lines 7 and 15. The principle behind the loop is to look at each character in the text variable c one at a time for a value of 13. The character we are looking at is determined by the for loop variable 'i'. A value of 13 indicates a 'newline' character. When we find a newline character we push the current value of str

into the categories array at line 9, then update an ID value and clear the string str. If the value is not 13 then the current character is added to the temporary string variable str. Having executed the loop we should now have an array of category labels stored in the global variable ‘_global.categories’.

Lines 16–27 use the contents of ‘_global.categories’ to set the UI components. Sprite channel 17 stores the categories stepper and so we need to set the minimum and maximum values for the stepper to allow users to move through the categories. Finally at line 25 we update the status text and set the value of ‘gCatID’ to –1, so that the loop does not repeat the same code for every loop. Notice the use of the function call ‘setCategory’. This function is one of several you will have noticed in the movie script.

Take a look at lines 86–93 of Listing 14.7. Here we first decrement the value of the passed parameter ‘id’. This is simply because the values in the steppers start at 1 while an array in JavaScript starts at zero. Lines 88–90 are a second call to ‘getNetText’. This time we are calling the ASP page ‘getCategory.asp’. The getCategory.asp page takes a single parameter ‘category = xxx’. ASP pages receive parameters using a querystring. You add a question mark symbol to the end of the URL then add your parameters in ‘name = value’ blocks, each parameter being separated from the previous using the ampersand symbol. The getCategory.asp page returns the text using the code value ‘gQuestionID’. This is the second section of the ‘enterFrame’ function.

The code is between lines 28 and 52 of Listing 14.8. Again we initialize a new array, this time called ‘questions’, at line 35. Lines 36–44 simply retrieve the value of the first line. The first line of the text contains the number of questions, allowing us to enter a ‘for’ loop to retrieve each question. Line 46 calls the movie script function ‘addQuestion’. This function takes two parameters, the text block and the position in the text block. The function is between lines 26 and 75 of Listing 14.7. A question is stored in a JavaScript object. Lines 27–33 simply create and initialize this object, lines 34–42 retrieve the difficulty level from the text, lines 41–47 get the actual question, lines 48–53 get the correct answer and lines 54–72 get the optional answers one at a time. Finally at line 73 the question object is pushed to the global variable questions.

Returning to the ‘enterFrame’ function, having placed all the questions in an easily accessed data format into the global variable questions, we first set the current category at line 48 and then set the question at line 49. This uses the movie script function ‘setQuestion’, which takes a single parameter, the index of the question. The function is listed between lines 77 and 84 of Listing 14.7. Here the values of the UI components are all updated from the questions array.

```

1  function enterFrame() {
2      if (netDone(_global.gCatID) && (netError(_global.gCatID) == "OK")) {
3          var c = netTextResult(_global.gCatID), str;
4          var id = 0;
5          _global.categories = new Array();
6          str = "";
7          for (var i=0; i<c.length; i++){
8              if (c.charCodeAt(i)==13){
9                  _global.categories.push(str);
10                 id++;
11                 str = "";

```

```

12         }else{
13             str += c.charAt(i);
14         }
15     }
16     if (_global.categories.length>0){
17         sprite(17).minimum = 1;
18         sprite(17).maximum = _global.categories.length;
19         sprite(17).value = 1;
20         setCategory(1);
21     }else{
22         sprite(17).minimum = sprite(17).maximum = 0;
23         sprite(25).minimum = sprite(25).maximum = 0;
24     }
25     member("status").text = "Category list loaded";
26     _global.gCatID = -1;
27 }
28 if (netDone(_global.gQuestionsID) &&
29 (netError(_global.gQuestionsID) == "OK")) {
30     var c = netTextResult(_global.gQuestionsID), str;
31     var id = 0;
32     var i = 0;
33     var total;
34     str = "";
35     _global.questions = new Array();
36     while (1){
37         if (c.charCodeAt(i)==13){
38             total = Number(str);
39             break;
40         }else{
41             str += c.charAt(i);
42         }
43         i++;
44     }
45     for (var n=0; n<total; n++){
46         i = addQuestion(c, i);
47     }
48     member("Category").text = _global.categories[sprite(17).value-1];
49     setQuestion(1);
50     member("status").text = "Category loaded";
51     _global.gQuestionsID = -1;
52 }
53

```

```

54     _movie.go(_movie.frame);
55 }

```

Listing 14.8

In addition to the scripts shown in Listings 14.7 and 14.8 the buttons and steppers all have behaviors that handle the 'click' event and the steppers the 'change' event. Take a look at 'Examples/Chapter14/createquiz.dir' to see these. They are all very simple scripts. The 'Enter question' button is the most complex because it needs to build a relatively complex querystring to pass to the 'addQuestion.asp' page.

Ideas for the implementation

At this stage you have an engine to enter and update quiz questions. The actual implementation of the quiz is left to the developer. You can access the questions based on category and difficulty. An interactive quiz should include some scaling of the difficulty and certain hurdles that the player must get over in order to move on to an extra life. Maybe you can offer some useful lifelines initially by reducing the number of possible answers or making suggestions that are often but not always correct. The quiz could be made funny by including a cartoon character that responds to correct and incorrect answers in an amusing way. The quiz could use a high-score table that offers prizes on a daily, weekly or monthly basis. Maybe the quiz could be part of an e-mail program, where

createquiz

Pub Quiz

Difficulty:

Question:

Correct Answer:

Option 1:

Option 2:

Option 3:

Category loaded:

Select Category:

Select Question:

General

Add Category

Delete category

To create or delete an entire category click the buttons above. To enter or amend a question click 'Enter question'. To delete a question click 'Delete question'. To select a new category use the 'Select Category' stepper. To clear the existing question just press 'Reset'. To move between questions in a category use the 'Select Question' stepper.

Delete question

Reset

Enter question

Figure 14.6 Using the input application

players could send 10 questions to a friend. However it is implemented, a quiz is a popular form of entertainment and easily updated. Because we have included the input date in the database we could easily offer the player the opportunity just to select from the new questions. This breathes new life into a website and encourages more visits.

Summary

Most of this chapter dealt with database access. This is a very important subject, because on even the simplest site you will probably want to collect some data from the users, even if it is only e-mail addresses and current high score. Director MX 2004 has all the tools necessary to link to a server to both load and save this data.

15 Platformers

Creating a platform-based game is quite a challenge, but by this stage in the book you should have all the necessary skills. In this chapter we are going to look at the problems of controlling the behavior of a sprite in a platform world. First we will look at an example where we move the sprite and the background remains stationary and then we will look at using a multi-layered scrolling background. Platform games often use dynamically created sprites; we will look at creating sprites on the fly and removing them when they are no longer in use.

The basics of platform games

Platform games involve two fundamental concepts. First, user control of a sprite character, which can usually perform several actions: walk, run, jump, fall and so on. Second, collision detection to ensure that the character remains anchored to the world that is presented to the player. Both parts of creating a platform game need careful coding to ensure success. You may prefer to start with the collision testing or with the user control; either way they are two separate problems and should be tested and debugged separately as far as possible. We will start with the problem of user control.

Responding to user input

Usually the only control the user has over the game play will be via a few key presses or via the mouse. Often the control is context sensitive. For example, if a character is falling then the player cannot set a new direction until the character has landed and the player regains control. For this reason always use a control flag. If the character is under player control then set the flag to 'true' and if the character is being controlled by the computer then set the control flag to 'false'. Because, wherever possible, we want to keep all the data pertaining to a sprite in one place, put the flag inside a sprite property list. The actions a sprite can perform are often limited both by its placement on screen and the action that is currently being performed, so for these reasons use a variable that stores the current action. Responding to keyboard input is easier to handle than mouse input for a platform game. Usually keyboard control will mean reading the arrow keys, space bar and possibly the control and shift keys. Reading the keyboard is easy using the `'_key.keypressed()'` construct. To clarify the principles let's look at an example. Open `'Examples/Chapter15/Platformer01.dir'`. Run the program. Using the arrow keys you can move the mouse character around the screen, jumping and landing on the various platforms that you can see on the screen.

A simple example

See Figure 15.1. The code for this example is mainly located in a behavior script for the mouse sprite. The script uses three property variables, which are initialized using the `'beginSprite'` handler.

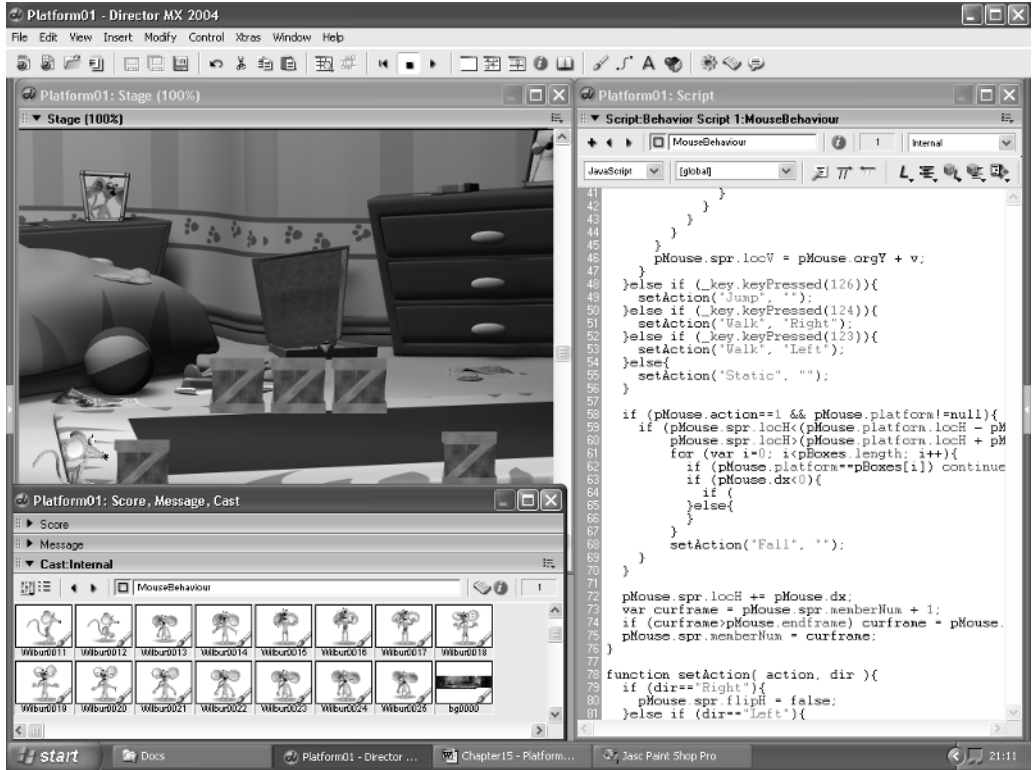


Figure 15.1 Developing 'Examples/Chapter15/Platformer01.dir'

The scripting code for this example is all written using JavaScript. Listing 15.1 shows the beginSprite handler. JavaScript can initialize an arbitrary 'Object'; this can be assigned member variables. This is similar to using a property list but is more flexible and dynamic. Essentially you can build a custom type that contains named variables of any type at runtime. The Object method is used to store all the information necessary to control the mouse sprite. In addition this small listing also initializes the platforms; the sprites that contain them are stored in an array, again similar to a Lingo list.

```

1  var pMouse, pBoxes, pJumpTime;
2
3  function beginSprite(){
4      pMouse = new Object();
5      pMouse.action = 0;
6      pMouse.spr = sprite(this.spriteNum);
7      pMouse.floorY = pMouse.spr.locV;
8      pMouse.dx = 0;
9      pMouse.dy = 0;
10     pMouse.platform = null;

```

```

11  pMouse.base = 87;
12  pBoxes = new Array(sprite(2), sprite(3), sprite(4),
13                      sprite(5), sprite(6));
14  setAction("Static", "");
15 }

```

Listing 15.1

The ‘enterFrame’ function

The main functionality is handled by the ‘enterFrame’ function. The full script for this function is given in Listing 15.2. It is a fairly complicated piece of code so let’s take it a bit at a time. First the enterFrame event is called by the system for each screen update. In other programming languages you would consider it a ‘callback’ function. What we need to know is that this function will be called however many times per second you have set the tempo for your movie. You need, therefore, to handle every condition that can occur in your movie within this code segment. Line 2 of Listing 15.2 shows a test for the mouse character, the data of which is stored in the property variable ‘pMouse’; the test is for when the ‘action’ is equal to 3; this happens when the mouse character is jumping. The block of code that follows is a simple projectile motion code segment, details of which are shown in the box that follows later in this chapter. Essentially you need to know that the position of the mouse vertically is handled by adding the origin value ‘pMouse.orgY’ to the frame-calculated value of ‘v’. This varies with time from ‘jumpTime’. So the mouse will jump up, decreasing the ‘y’ value on screen, until a certain time when it will start to fall. If the value of combining the current origin and the value for v for the current frame places the mouse below the floor level then we can be assured that the mouse has landed and so in line 8 we set the vertical value for the mouse to the floor level and either start the mouse walking again or stop it depending on the value of ‘pMouse.dx’.

The ‘dx’ variable defines the horizontal movement. A negative value for dx will have the mouse walking left and a positive value walking right. A value for dx of zero means the mouse is static. Notice that the current action of the mouse is set by a function call in lines 10 and 12. We will examine the code for this function a little later. If the test at line 7 for the mouse being below the floor level fails, which in most cases on a jump will be the case, then the code block between lines 15 and 36 is executed. First we test whether the mouse is falling. This will occur when the current position of the sprite is updated to a higher value by setting the value to the origin y value plus the current value for v. This is the test used in line 15.

We can only have a landing on a box when the mouse is falling. We could also check for hitting a box in the horizontal direction and then slide down the side, for example; in this simple example we are only checking the vertical case in an attempt to make the code less confusing. If you are hopelessly confused by the code then I apologize, but try and work through it; with a little perseverance you will find that it is more logical than it may at first appear. Be aware that collision testing is always mode conditional in a 2D computer game. In this instance we only check for the tops of boxes when the mouse is falling. If the mouse lands on top of a box then we need to set its new vertical position and inform the code that we are now on a box rather than on the floor. A box has a horizontal limit, which we need to test. If this limit is exceeded then we need to initiate a fall.

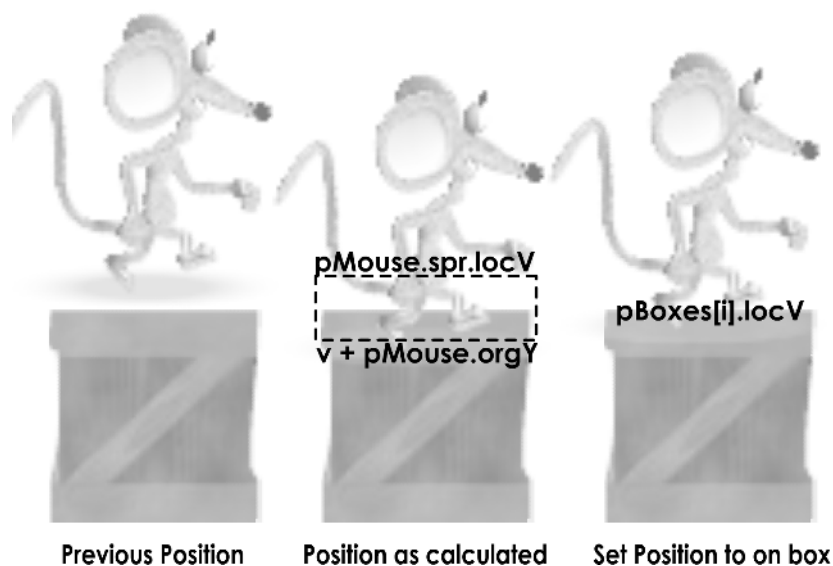


Figure 15.2 *Legal landing areas*

So how do we test for a land? This occurs when the mouse's current position is above the location of the box and the vertical position we intend to set is below the box. If this is the case then the tests on lines 19 and 20 will evaluate to true, hence the code block between 21 and 35 will be executed. The next test is whether the mouse's horizontal position is within the horizontal limits of the box. This occurs when the mouse sprite's horizontal position, 'locH', is greater than the horizontal position of the box minus half its width and less than the horizontal position of the box plus half its width. If this is the case then the tests on lines 21 and 22 will evaluate to true and the code block between lines 23 and 30 will be used. Here we set the value of 'platform' for the 'pMouse' property variable to point to the box on which the mouse has landed. We also set a 'land' flag to true so we can use this later in the code block. On line 25 we set the value of the mouse's vertical position to the top of the box. This is an important step. In platform games you will want your sprites to land at known vertical locations. Always hard set these, otherwise you will find that, particularly after a long fall, the sprite will intersect the platforms or hover above them, whereas the feet matching the platform top is the intention in the game. Line 35 uses the land flag. If no land has occurred then we set 'locV' for the sprite and continue falling. If a 'land' has taken place then this code line will be ignored since the vertical location has been hard set at line 25. The land relies on the registration points for the mouse sprite and the box being located as shown in Figure 15.2. The registration point for the mouse is at the bottom of the bitmap between the mouse's feet while the registration point for the box is at the mid-point of the top line of the box. The image on the right of Figure 15.3 shows the registration points for the two bitmaps set to the same location. The locV and locH values are all relative to this registration point. The registration point can be set with the 'Paint' window or in Lingo. The position once set is persistent in the cast. It is useful to use

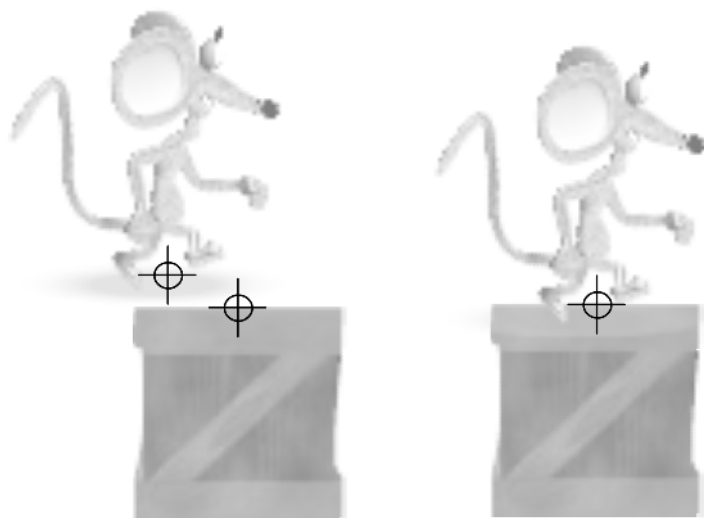


Figure 15.3 *Registration points*

the code snippet

```
for (i=2; i<11; i++){
    member(i).regPoint = member(1).regPoint;
}
```

This would set the registration point of the cast members 2–10 to match the registration point of member 1. In this way you can set a block of bitmaps to an accurate registration point. Once set the code block can be omitted because the registration points will be remembered.

The 'enterFrame' function continues at line 37 where if the mouse is not currently jumping we start one of three keyboard tests. We test for the left, right and up keys. Each one uses a call to the function 'setAction'. If the mouse is not jumping and none of the keys are pressed then we set the current action to static at line 44. Lines 47–59 are only executed when the mouse is walking and the mouse is standing on a platform when the value of 'pMouse.platform' is not null. We test for the limit of the platform in lines 48–50. If the limit is passed then we set the mouse's action to 'Fall'. Finally we do a little housekeeping in lines 56–59. First we update the horizontal location based on the value of the variable 'dx'. Then we update the displayed frame for the sprite. If this exceeds a previously set value for 'endFrame' then we loop back to 'startFrame'. In line 59 we set the value of the cast member for this sprite to the value calculated as 'curframe'.

```
1 function enterFrame( me ){
2     if (pMouse.action==3){
3         //No keyboard control if jumping or falling
4         //Simple projectile motion
5         t = (_system.milliseconds - jumpTime)/100.0;
```

```

6      v = t * t * 10.0 - t * 60.0;
7      if ((pMouse.orgY + v)>pMouse.floorY){
8          pMouse.spr.locV = pMouse.floorY;
9          if (pMouse.dx!=0){
10             setAction("Walk", "");
11         }else{
12             setAction("Static", "");
13         }
14     }else{
15         if ((pMouse.orgY + v)>pMouse.spr.locV){
16             //Going down so test for a land
17             var land = false;
18             for (var i=0; i<pBoxes.length; i++){
19                 if ((pMouse.orgY + v)>pBoxes[i].locV &&
20                     pMouse.spr.locV<pBoxes[i].locV){
21                     if (pMouse.spr.locH>(pBoxes[i].locH - pBoxes[i].width/2) &&
22                         pMouse.spr.locH<(pBoxes[i].locH + pBoxes[i].width/2)){
23                         pMouse.platform = pBoxes[i];
24                         land = true;
25                         pMouse.spr.locV = pBoxes[i].locV;
26                         if (pMouse.dx!=0){
27                             setAction("Walk", "");
28                         }else{
29                             setAction("Static", "");
30                         }
31                     }
32                 }
33             }
34         }
35         if (!land) pMouse.spr.locV = pMouse.orgY + v;
36     }
37 }else if (_key.keyPressed(126)){
38     setAction("Jump", "");
39 }else if (_key.keyPressed(124)){
40     setAction("Walk", "Right");
41 }else if (_key.keyPressed(123)){
42     setAction("Walk", "Left");
43 }else{
44     setAction("Static", "");
45 }
46
47 if (pMouse.action==1 && pMouse.platform!=null){

```

```

48     if (pMouse.spr.locH < (pMouse.platform.locH - pMouse.platform.width/2)
49         || pMouse.spr.locH >
50             (pMouse.platform.locH + pMouse.platform.width/2)) {
51         setAction("Fall", "");
52     }
53 }
54 }
55
56 pMouse.spr.locH += pMouse.dx;
57 var curframe = pMouse.spr.memberNum + 1;
58 if (curframe > pMouse.endframe) curframe = pMouse.startframe;
59 pMouse.spr.memberNum = curframe;
60 }

```

Listing 15.2

The ‘setAction’ function

To change from one action to another the ‘enterFrame’ function uses the ‘setAction’ function. The advantage of using a function to do the work is that you can handle all the aspects of changing mode in a single place. In a full game you are likely to call ‘setAction’ from a number of different places. If all the necessary changes involved in a new mode are handled in each of the different places in code then errors are likely to creep into your scripts. The setAction function is passed two parameters. The first is ‘Action’; the chosen new action is passed as a string value. For peak performance this could be a numeric value but for convenience of reading the code this is a string. The second parameter, ‘dir’, defines the direction of the mouse. Possible values are ‘Left’, ‘Right’, ‘Turn’ or an empty string. We use this parameter to set the current value for the sprite’s ‘flipH’, which dictates whether the sprite is flipped in the horizontal direction.

Following the code block associated with direction is a long code block that is a series of ‘if’ statements to handle the possible actions to be set. When setting the action to ‘Walk’ between lines 15 and 25 we first test to see if we are currently walking; this occurs when the ‘pMouse.action’ value is 1. If this is the case then we need do nothing so we return at line 15. If we are not currently walking then we set the pMouse.action value to 1. An action is associated with a block of animation frames. These are relative to the first frame in the cast of the mouse, making it easier to move the mouse frames around in the cast. ‘pMouse.base’ stores the value of the first mouse frame, ‘pMouse.startframe’ defines the first frame of the animation loop and ‘pMouse.endframe’ stores the last frame. When the mouse is walking, the ‘pMouse.dx’ value is ± 4 based on the value of ‘pMouse.spr.flipH’. When walking, the ‘pMouse.dy’ value is always zero. The final line of the block, line 25, sets the first animation frame. The jump action block between lines 27 and 41 is a little different from the others. In this one we store the ‘y’ location of the base of the jump and the time of the jump. We clear the platform because if the mouse has jumped it cannot be on a platform. Also we reduce the pMouse.dx value by 1, either by adding 1 or subtracting 1 depending on whether the value is greater or less than zero. These additional values stored in the property variable will be

used by the code block in the 'enterFrame' function to set the vertical position of the mouse. The fall block between lines 43 and 49 is similar to the jump block, the difference being the time set for the jump. This is back set by 600 milliseconds. The code block for a jump will cause the mouse to jump up before falling; the fall below the base line will occur at exactly 600 milliseconds after the start of the jump. By setting the time in the way shown on line 48, we are immediately falling, which is the desired result. To fall from zero speed you could also subtract the height of the jump from the current location and set this to be 'orgY' at line 46. See the box for more information on simple projectile motion.

The final code block sets up a static action; here both the start and end frames are the same and the movement values in the 'x' and 'y' directions are set to zero.

```

1  function setAction( action, dir ){
2      if (dir=="Right"){
3          pMouse.spr.flipH = false;
4      }else if (dir=="Left"){
5          pMouse.spr.flipH = true;
6      }else if (dir == "Turn"){
7          if (pMouse.spr.flipH){
8              pMouse.spr.flipH = false;
9          }else{
10             pMouse.spr.flipH = true;
11         }
12     }
13
14     if (action=="Walk"){
15         if (pMouse.action == 1) return;
16         pMouse.action = 1;
17         pMouse.startframe = 0 + pMouse.base;
18         pMouse.endframe = 11 + pMouse.base;
19         if (pMouse.spr.flipH) {
20             pMouse.dx = -4;
21         }else{
22             pMouse.dx = 4;
23         }
24         pMouse.dy = 0;
25         pMouse.spr.memberNum = pMouse.startframe;
26     }else if (action=="Jump"){
27         if (pMouse.action == 3) return;
28         pMouse.startframe = 0 + pMouse.base;
29         pMouse.endframe = 3 + pMouse.base;
30         pMouse.orgY = pMouse.spr.locV;
31         pMouse.platform = 0;
32         jumpTime = _system.milliseconds;

```

```

33     switch(pMouse.dx) {
34     case -4:
35         pMouse.dx++;
36         break;
37     case 4:
38         pMouse.dx--;
39         break;
40     }
41     pMouse.action = 3;
42 }else if (action=="Fall"){
43     if (pMouse.action == 3) return;
44     pMouse.startframe = 0 + pMouse.base;
45     pMouse.endframe = 3 + pMouse.base;
46     pMouse.orgY = pMouse.spr.locV;
47     pMouse.platform = 0;
48     jumpTime = _system.milliseconds - 600;
49     pMouse.action = 3;
50 }else if (action=="Static") {
51     if (pMouse.action == 2) return;
52     pMouse.action = 2;
53     pMouse.startframe = 12 + pMouse.base;
54     pMouse.endframe = 12 + pMouse.base;
55     pMouse.dx = 0;
56     pMouse.dy = 0;
57     pMouse.spr.memberNum = pMouse.startframe;
58 }
59 }

```

Listing 15.3

Walking along a line of boxes

If you try the example 'Examples/Chapter15/Platformer01.dir' then you will see that there is a problem. When the mouse walks along a line of boxes it falls at the end of the first box. Listing 15.4 shows a fix for this, which is added to the 'enterFrame' function. We test for the end of a platform in the usual way. If a fall is required then we test all the boxes to see if they are within 10 pixels of the current location; if they are then we swap the platform support from the platform we have just reached the end of to the new platform. The test varies depending on whether we are walking left or right.

```

1  if (pMouse.action==1 && pMouse.platform!=null){
2      if (pMouse.spr.locH<(pMouse.platform.locH - ↵
          pMouse.platform.width/2) ||
3          pMouse.spr.locH>(pMouse.platform.locH + ↵
          pMouse.platform.width/2)) {

```

```

4      trace("Walking on platform " + pMouse.platform);
5      var fall = true;
6      for (var i=0; i<pBoxes.length; i++){
7          if (pMouse.platform==pBoxes[i]) continue;
8          if (pMouse.dx<0){//Walking left
9              if (pBoxes[i].locH < pMouse.platform.locH &&
10                 (pMouse.platform.locH - pBoxes[i].locH) <
11                 (pMouse.platform.width + 10)){
12                  pMouse.platform = pBoxes[i];
13                  trace("Overlap left " + pBoxes[i]);
14                  pMouse.locH = pMouse.platform.locH +
15                             pMouse.platform.width/2 - 2;
16                  fall = false;
17                  break;
18              }
19          }else{//Walking right
20              if (pBoxes[i].locH > pMouse.platform.locH &&
21                 (pBoxes[i].locH - pMouse.platform.locH) <
22                 (pMouse.platform.width + 10)){
23                  pMouse.platform = pBoxes[i];
24                  trace("Overlap right " + pBoxes[i]);
25                  pMouse.locH = pMouse.platform.locH -
26                             pMouse.platform.width/2 + 2;
27                  fall = false;
28                  break;
29              }
30          }
31      }
32      if (fall) setAction("Fall", "");
33  }
34 }

```

Listing 15.4

Simple projectile motion

Instead of using lots of 'if ... then' structures for a jump or fall we can use a simple physics calculation. The y value for a projectile motion is defined as

$$y = ut + gt^2,$$

where u is the launch value, t is the time in seconds and g is the value of gravity. If t is less than 1 s the value for t^2 will be less than t ; but above 1 s, t^2 will overtake. If g operates in the opposite direction to u

then the increase in the values for y will decrease and eventually reverse. But this sounds all very hit and miss. Actually it is easy to calculate. Suppose that you want a jump to occur over 1 s and the starting and ending values for y are zero; then you want

$$ut = -gt^2,$$

when $t = 0$ and $t = 1$. To do this you need to solve a quadratic equation. Whoops, that sounds like maths to me! Believe me, it is not that hard. You want

$$ut + gt^2 = 0.$$

But we also want to control the maximum height of the jump. This occurs when the derivative of the curve has a value of zero. The derivative of the curve is

$$u + 2gt$$

OK so let's fiddle with the figures. If we want a maximum height for the curve of 150 and if the total jump takes 1 s then the maximum height is going to be reached at 0.5 s; half way through in other words. At $t = 0.5$ we want

$$u + 2gt = 0 \text{ and } ut + gt^2 = 150.$$

Replacing t with 0.5 gives

$$u + g = 0 \text{ and } 0.5u + 0.25g = 150.$$

Two equations with two unknowns can be solved. From the first equation we know that $u = -g$, therefore we can substitute u for $-g$ in the second equation giving $-0.5g + 0.25g = 150$ or $-0.25g = 150$, which means that $g = -600$ and $u = 600$.

If you are working in meters rather than pixels then the usual value for g is -9.81 ms^{-2} ; for convenience we will say -10 ms^{-2} . Imagine that the mouse is about 1 m high and can jump just its own height. Therefore

$$u - 20t = 0 \text{ and } ut - 10t^2 = 1.$$

From the first equation we know that $u = 20t$. We can put this into the second equation and see that

$$20t^2 - 10t^2 = 1 \text{ or } t^2 = 1/10 \text{ or } t \text{ is approximately } 0.32 \text{ s.}$$

So the peak of the curve will occur at around about one-third of a second.

If you are able to follow this then it is without doubt the best way to get the curved motions that you want. But if it is all totally baffling then just try changing the values until you get the result you

want. Trial and error never hurt. Physics can, however, be a very useful way of creating very smooth animations and it is well worth trying to get your head around some simple manipulation of equations. You are unlikely to need more than some simple trigonometry, a little basic algebra and enough calculus to understand how to work out the derivative of a curve when working with Director games.

To work out the derivative of a polynomial, you simply multiply the coefficient by the power and reduce the power by 1. Therefore x^2 becomes $2x$, $4x^3$ becomes $3 \cdot 4x^{(3-1)} = 12x^2$, $5x$ becomes just 5 and a constant drops out to zero. Consequently the derivative of

$$4x^3 + x^2 + 5x + 3$$

is

$$12x^2 + 2x + 5.$$

The derivative gives the slope of a curve. The maximum or minimum of a quadratic curve (a curve where the highest power is 2) occurs when the derivative has the value zero because this indicates that the slope of the curve at this point is horizontal.

Lecture over, let's make some games!

Using a scrolling background

Most players expect a platform game to scroll. On a scroller the main character stays central and everything else moves. We can use most of the code from the previous example with a few exceptions. Open 'Examples/Chapter15/Platformer03.dir'. Take a look at the game first so you can see how it behaves. Notice that we have what is called parallax scrolling; that is, things further away scroll less than things nearer to the camera. To achieve this, the initial parameters are set in the 'beginSprite' handler. This example uses Lingo scripting, so you can choose between your preferred scripting languages. We start by assigning a few convenience variables to point to the background and foreground sprites, the mouse and all its data and the base vertical position of both the background and the foreground, the 'locV'. The 'enterFrame' handler is called repeatedly and works in almost the same way as the previous version. The difference is the mouse stays static and the background moves. Look at lines 28 and 29 of Listing 15.5. Here we move the 'bg' based on the calculation for 'v' but the foreground moves 1.2 times this amount. This is to reproduce the parallax effect; things closer to the viewer move more than things further away. Notice also at lines 31 and 32 that the same increase is used this time by multiplying the value for the mouse's movement horizontally stored in the property list variable 'dx'. Adding parallax scrolling greatly enhances the feeling of depth.

```
1 property bg, foreground, mouse, bgY, foreY, jumpTime
2
3 on beginSprite me
```

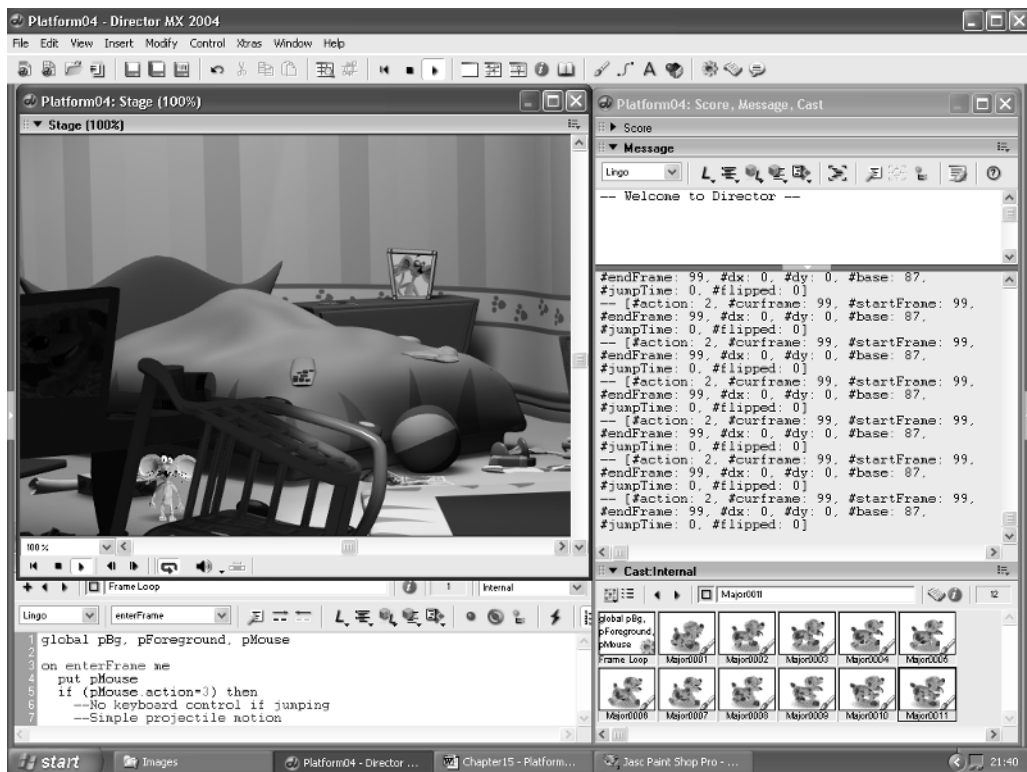


Figure 15.4 Testing 'Examples/Chapter15/Platformer04.dir'

```

4  bg = sprite(1)
5  foreground = sprite(3)
6  mouse = [#spr:sprite(2), #action:1, #startframe: 0, #endframe:11, \
7          #dx:0, #dy:0, #base:member("Wilbur0001").number]
8  bgY = bg.locV
9  foreY = foreground.locV
10 setAction("Static", "")
11 end
12
13 on enterFrame me
14   if (mouse.action=3) then
15     --No keyboard control if jumping
16     --Simple projectile motion
17     t = (_system.milliseconds - jumpTime)/100.0
18     v = t * 60.0 - t * t * 10.0
19     if (v<0) then
20       bg.locV = bgY

```

```

21     foreground.locV = foreY
22     if (mouse.dx<>0) then
23         setAction("Walk", "")
24     else
25         setAction("Static", "")
26     end if
27 else
28     bg.locV = v + bgY
29     foreground.locV = v * 1.2 + foreY
30 end if
31 bg.locH = bg.locH - mouse.dx
32 foreground.locH = foreground.locH - mouse.dx * 1.2
33 else if (_key.keypressed(" ")) then
34     setAction("Jump", "")
35 else if (_key.keypressed(126)) then
36     setAction("Walk", "Right")
37     bg.locH = bg.locH - mouse.dx
38     foreground.locH = foreground.locH - mouse.dx * 1.2
39 else if (_key.keypressed(125)) then
40     setAction("Walk", "Left")
41     bg.locH = bg.locH - mouse.dx
42     foreground.locH = foreground.locH - mouse.dx * 1.2
43 else
44     setAction("Static", "")
45 end if
46
47 curframe = mouse.spr.memberNum + 1
48 if (curframe>mouse.endframe) then curframe = mouse.startframe
49 mouse.spr.memberNum = curframe
50 end

```

Listing 15.5

Painting the scene directly using ‘copyPixels’

Depending on your computer you may find that the previous example had a slower than preferred frame rate. One way to speed things up is to draw your sprites with your own code, rather than rely on sprites on the score. To do this you will use the ‘copyPixels’ method. Most of the functionality is placed in a frame script on frame 1 of the score. The script is shown in Listing 15.6. The script makes use of global property lists that are initialized by the ‘startMovie’ handler in the movie script for this movie. Much of the script will be familiar. Instead of using the ‘locH’ and ‘locV’ properties of a sprite, we are tracking these ourselves using the ‘x’ and ‘y’ properties of the ‘gBg’ and ‘gForeground’ property lists. Take line 10, for example, which sets the value of ‘gBg.y’. The significant difference in this script is from line 41 to the end of the code. Here we set up and use the

copyPixels method. The copyPixels method takes up to four parameters and it is a method of an image object. So we need an image object to use it. Because our drawing is going directly to the stage we use the image object of the stage. The first parameter is the source of the pixels we are copying; in the example this is a cast member. The second parameter is the target rectangle for the copy. Notice that at line 41 we create a rectangle that is the full size of the stage. The parameters for the 'rect' function are left, top, right and bottom in that order and are measured in pixels. So line 41 creates a rectangle with left and top equal to zero, right set to 550 and bottom set to 400. Line 42 is slightly more complex. The copyPixels method can be used to stretch an image. In our example we are not doing any stretching so the sizes of the source and destination rectangles need to be the same. However the position of them does not need to be the same. So we are aiming at two rectangles where right minus left are equal and bottom minus top are equal, but all the values can differ as long as the 'size' of the rectangles matches. In line 42 we take the current x and y values for gBg and use these as the left and top values; therefore the right and bottom values must be the same values plus the width and height of the rectangle, giving us the code in line 42. Now we can copy the appropriate bit of the background for the current frame to the stage. This effectively wipes out the previous frame because it paints the complete stage area. It is important to do this if you are using copyPixels because otherwise you will leave a trail of pixels behind.

Lines 45–48 calculate the 'src' and destination rectangles for the mouse and paint it to the stage. Finally lines 50–53 look after the painting of the foreground element. You will find the performance of this to be greatly improved over the score version. But it is a little more complicated to handle; remember to paint the sections starting from the back because each paint goes on top of the previous one.

```

1  global gBg, gForeground, gMouse
2
3  on enterFrame me
4    if (pMouse.action=3) then
5      --No keyboard control if jumping
6      --Simple projectile motion
7      t = (_system.milliseconds - gMouse.jumpTime)/100.0
8      v = t * t * 10.0 - t * 60.0
9      if (v>0) then
10         gBg.y = gBg.orgY
11         gForeground.y = gForeground.orgY
12         if (pMouse.dx<>0) then
13           setAction("Walk", "")
14         else
15           setAction("Static", "")
16         end if
17       else
18         gBg.y = v + gBg.orgY
19         gForeground.y = v * 1.2 + gForeground.orgY
20       end if

```



```

21     gBg.x = gBg.x + gMouse.dx
22     gForeground.x = gForeground.x + gMouse.dx * 1.2
23     else if (keypressed(126)) then
24         setAction("Jump", "")
25     else if (keypressed(124)) then
26         setAction("Walk", "Right")
27         gBg.x = gBg.x + gMouse.dx
28         gForeground.x = gForeground.x + gMouse.dx * 1.2
29     else if (keypressed(123)) then
30         setAction("Walk", "Left")
31         gBg.x = gBg.x + gMouse.dx
32         gForeground.x = gForeground.x + gMouse.dx * 1.2
33     else
34         setAction("Static", "")
35     end if
36
37     gMouse.curframe = gMouse.curFrame + 1
38     if (gMouse.curframe>gMouse.endframe) then
39         gMouse.curframe = gMouse.startframe
40
41     destrect = rect(0, 0, 550, 400)
42     srcrect = rect(gBg.x, gBg.y, gBg.x + 550, gBg.y + 400)
43     _movie.stage.image.copyPixels(gBg.img, destrect, srcrect)
44
45     img = member(gMouse.curframe).image
46     srcrect = rect(0, 0, img.width, img.height)
47     destrect = rect(100, 300, 100 + img.width, 300 + img.height)
48     _movie.stage.image.copyPixels(img, destrect, srcrect)
49
50     destrect = rect(0, 0, 550, 400)
51     srcrect = rect(gForeground.x, gForeground.y, gForeground.x + 550, \
52         gForeground.y + 400)
53     _movie.stage.image.copyPixels(gForeground.img, destrect, srcrect)
54
55     _movie.go(_movie.frame)
56 end

```

Listing 15.6

Creating sprites dynamically

Very often in a platform game you will want to create sprites dynamically. Maybe your central character hits a box and stars fly out, for example. It is quite easy to create sprites in this way using a Lingo technique called parent scripting. This technique is rather like classes in JavaScript and can

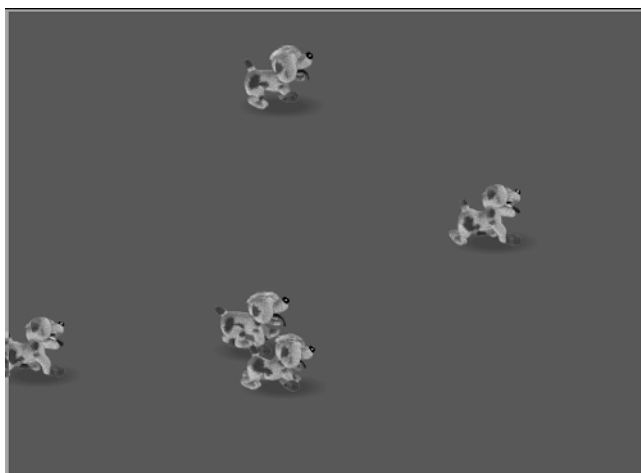


Figure 15.5 *Dynamically created sprites using 'new'*

even use inheritance; one script can have an ancestor and so inherits all the attributes of the parent on which the new script builds. This methodology called object-orientated programming is the preferred model for most programmers and is the foundation of both Java and C++. In this simple example we are just going to introduce the idea of it; it is very easy to use and acts as a useful technique particularly in a dynamic game that must create and kill many sprites over the duration of the game play.

Listing 15.7 shows how a script can be initialized using 'new'. You pass as a parameter the member name of the script and within the parentheses following new you place any parameters necessary for the initialization. In the example at line 6 there is a single parameter but in your code there can be as many parameters as you choose, separated by the usual commas.

```

1  global gBuckets
2
3  on startMovie
4    gBuckets = [0, 0, 0, 0, 0]
5    repeat with i = 1 to 5
6      gBuckets[i] = script("Bucket").new(i)
7    end repeat
8    put gBuckets
9  end

```

Listing 15.7

When you call new for the script it looks for a new handler. Listing 15.8 shows the full script for the member 'Bucket'. Notice the new handler between lines 3 and 11. This is passed the parameter following the 'me' parameter. All Lingo handlers receive me as the first parameter, so the first parameter you use will actually be the second in the function definition. In the new function we

set up the property variables for this script, giving a starting location, a frame and a speed and most importantly returning me to the caller. Only by returning me can the assignment in line 6 of Listing 15.7 take place. The update handler simply paints the new sprite frame using the ‘copyPixels’ method after setting the new value for ‘x’ and frame.

```

1  property id, x, y, frame, speed
2
3  on new(me, i)
4    put "new " & i
5    id = i
6    x = random(550)
7    y = random(300)
8    frame = random(14)
9    speed = random(5)
10   return me
11 end
12
13 on update
14   img = member(frame).image
15   srcRect = rect(0, 0, img.width, img.height)
16   destRect = rect(x, y, x + img.width, y + img.height)
17   _movie.stage.image.copyPixels(img, destRect, srcRect)
18   frame = frame + 1
19   if frame > 14 then frame = 1
20   x = x + speed
21   if x > 550 then x = -50
22 end

```

Listing 15.8

Left like this nothing will happen; you need to create a loop script that repeatedly calls the scripts update method. Listing 15.9 shows the script you need. ‘Examples/Chapter15/Platformers05.dir’ shows this example in action.

```

1  global gBuckets
2
3  on enterFrame()
4    _movie.stage.image.fill(0, 0, 550, 400, rgb(0, 150, 0))
5    repeat with i=1 to gBuckets.count
6      gBuckets[i].update()
7    end repeat
8    _movie.go(_movie.frame)
9  end

```

Listing 15.9

Summary

Platform games can be quite difficult to code but if you are methodical and prepared to rigorously test each part of your script then they can be great fun to produce and certainly great fun to play. The examples in this chapter give you the basis for a game. Hopefully you will create a functional game using these hints as a template; if you do then please send a URL of the game to me at nik@niklever.net. I look forward to seeing the results. I hope to create a page of readers' examples at www.niklever.net/director.

This page intentionally left blank

Section 4

The third dimension

The best thing about Director MX 2004 for the game developer is the 3D features. The final section takes the reader through the information needed to create their own 3D game.

This page intentionally left blank

16 Creating low-polygon characters

Before you can use Director's extensive 3D Xtra you will need to be familiar with at least one of the many 3D modeling packages. Director acts as the development platform where you add the interactivity; you cannot create the 3D elements that you use in the games using Director. This chapter provides an overview into the modeling process to help you choose which modeling package to use and how you can take the content created in the package and turn it into something that you can use with Director MX 2004. As such it extends on the ideas we met in Chapter 5.

Modeling software

At the time of writing, January 2004, the leading computer-generated imagery (CGI) packages are Maya, 3DS Max and Lightwave 3D. I choose to illustrate the modeling process using my preferred software, Lightwave 3D. Lightwave 3D is a polygon modeler, unlike Softimage, which in the latest version has no polygon tools at all. In fact it ships with the old version in order to provide polygon modeling tools. The other reason for choosing Lightwave 3D is the excellent developer support that is available free via the mailing list. See the back of the book for useful sources of support. The final reason for choosing Lightwave 3D is the fact that the file formats for models and animation are public domain and the available documentation is both comprehensive and accurate. Some CGI software developers have in recent years kept their file formats under wraps unless you become part of a very expensive developer network.

Choosing a modeling package

Following the examples in this chapter we are going to create two characters. One is the typically low-polygon superwoman, where an alarming amount of polygons are devoted to the upper body! The other is a very cartoony character. As an animator of some 20 years' experience I think it is disappointing that so much computer animation attempts to create a realistic interpretation of the real world, some more successfully than others. Animation allows the creators to develop fantasy lands that no one will ever experience any other way. As a developer I encourage you to develop content that provides a personal view of the world rather than a faithful recreation of reality. But, with a view to sales I have chosen to take the well-trodden route of the sexy girl.

Modeling the head

When creating our model we are aiming at around 1000 polygons for the central character. The supplied software gets 30 frames per second from 5000 poly scenes on an 800 MHz machine with a GeForce 2 card. When creating on-line 3D games for clients such as Kellogg's a P800 is our

base-level target machine. In no time at all that spec will seem ludicrously low but you will learn in later chapters how to scale your models for higher end platforms using subdivision surfaces. Lightwave 3D has an option to use subdivision surfaces but the algorithm used is slightly different from that used in Director. If you do work through the tutorial in the Lightwave package then by all means use the 'TAB' key to get a smoother subdivided mesh, but bear in mind that this smoother mesh is inside the cage rather than sitting on the cage. This has the effect of making particularly the limbs of the character appear slimmer than they will appear if you choose to display either the actual geometry modeled or a subdivided mesh based on this geometry but using an interpolating algorithm. The subdivision result that you will see in Director is similar but not an exact match to the one you will see in Lightwave 3D.

The principal tools we will use to create the mesh are detailed below.

- *Point creation tool* – using the point creation tool and the three standard views, top, front and side, you can create a single vertex in 3D space.
- *Polygon creation tool* – by selecting points and then choosing this tool a polygon is created with the selected vertices.
- *Weld* – any vertex can be welded to any other vertex. An extension of this for the Lightwave 3D modeler is Multiweld, which is provided on the CD. Multiweld is an LScript plugin for Lightwave that welds points based on their selection order. The first point selected is welded to the second point selected; the third to the fourth and in general the points with an odd index value are welded to those of an even index value, the index being based on selection order. This technique is useful for joining heads to bodies, legs to hips and arms to shoulders.
- *Drag* – this tool allows you to move a single point or a line of points.
- *Move* – this tool moves the current point or polygon selection.
- *Rotate* – this tool rotates the current point or polygon selection
- *Smooth shift* – this tool creates new geometry by taking a selected polygon(s) and extruding it along the vertex normals by an amount specified.
- *Bevel* – this tool insets a polygon(s) by a specified amount.
- *Knife* – this slices through the geometry creating new edges and dividing polygons where they have been sliced.
- *Magnet* – by defining an area the user can move a set of points with falloff.

We will look in more detail at each tool as we use it for the first time.

The importance of drawing

A sketch is very useful when creating low-polygon models. By using the sketch as a reference throughout the modeling process, the relative scale of the polygons you use to define your character is constantly available. The more accurate your drawing the easier you will find the modeling process. I used the sketches shown in Figure 16.1 as a backdrop for the front and side views when modeling Charlie.

In the sketch the pencil suggests the stretching of the fabric of the costume. In low-polygon modeling you make no attempt to recreate these creases with your geometry; this detail will be left

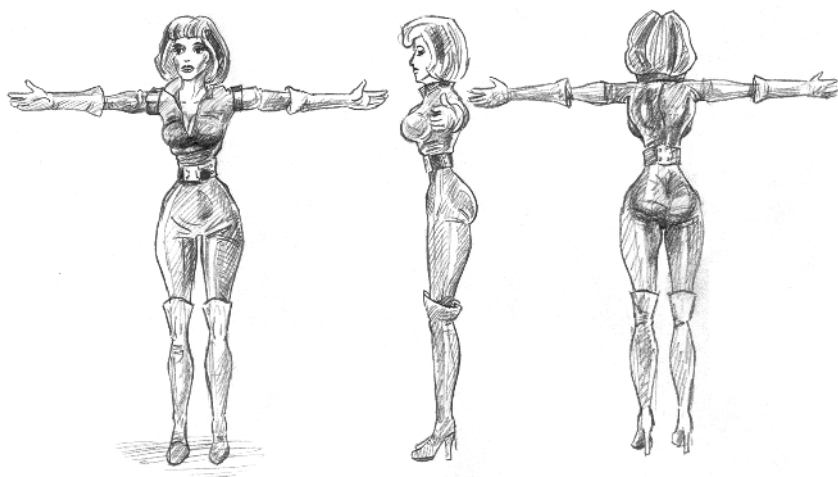


Figure 16.1 *Front, side and back view sketches of Charlie*

for texture mapping. The aim of your modeling is to define the main volume of the character. You might find it useful while modeling to add some relevant surface colors to your polygons to help when judging the results.

Triangles or quads

Ultimately your character is going to be a mesh that deforms as the character goes through her paces. As she deforms her vertices, any polygons that have more than three vertices will become nonplanar. Figure 16.2 shows the problems associated with nonplanar polygons. The polygon at the top left appears to be planar. As it rotates, however, it is clear that it is far from planar. This situation causes the render engine great difficulty in determining whether a polygon is front or back facing. The same diagram illustrates the polygon split into two triangles. Now it is clear how the geometry should be rendered.

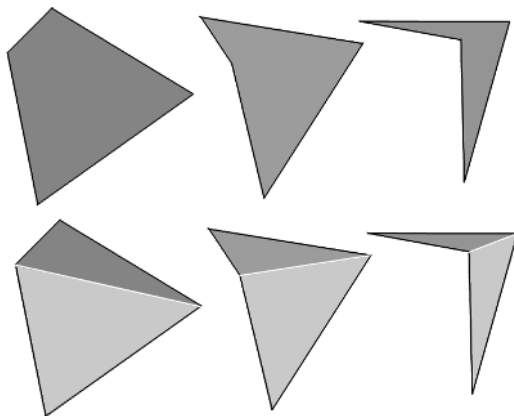


Figure 16.2 *The problems of nonplanar polygons*

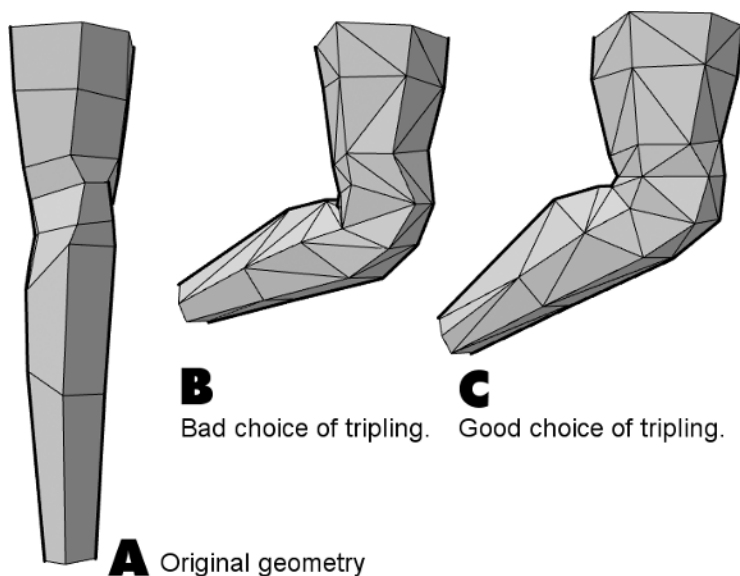


Figure 16.3 *How the tripling of quads can affect the rendering of a deformable mesh*

The problem of nonplanar polygons is that the rendering software will be considering only triangles. If you have a four-sided polygon then the renderer will effectively split this into two triangles. If one of these triangles is angled in such a way as to appear to be back facing, then your model will develop holes. For this reason your mesh must be triangular. But modeling with a triangular mesh is difficult. It is always easier to model with a combination of quads and triangles. So what should you do about those quads? All CGI packages have the ability to change a mesh into a triangular mesh, but this is done without regard to the overall geometry. In low-polygon models the direction in which a quad is split into two triangles makes a great difference to how the deforming mesh will appear when animating. There are intelligent triple engines available that attempt to put the edge where you would choose. No intelligent triple will do as good a job as an experienced artist, but when time is precious there is often no alternative. The areas of particular concern with respect to the way a quad mesh is tripled are where the mesh will be bent the most. These areas of maximum deformation can often be treated independently. If a mesh contains 1000 triangles then it is probably important to make sure that around 100 of these are tripled correctly. Around the armpit, elbow and knee are the key areas. Polygon modeling software will have the ability to make a new polygon from a point selection and then remove the old polygons that have the wrong orientation. Figure 16.3 shows what can happen around a knee joint if the polygon tripling chooses the wrong diagonal for the division. Good tripling can make an enormous difference to the way the silhouette of a mesh appears when deforming.

Making a start with the head

All modelers find their own methods. Some like to start with a ball or a box and deform it into the shape they are aiming at. Others like to start with a blank sheet. I choose one method for the head

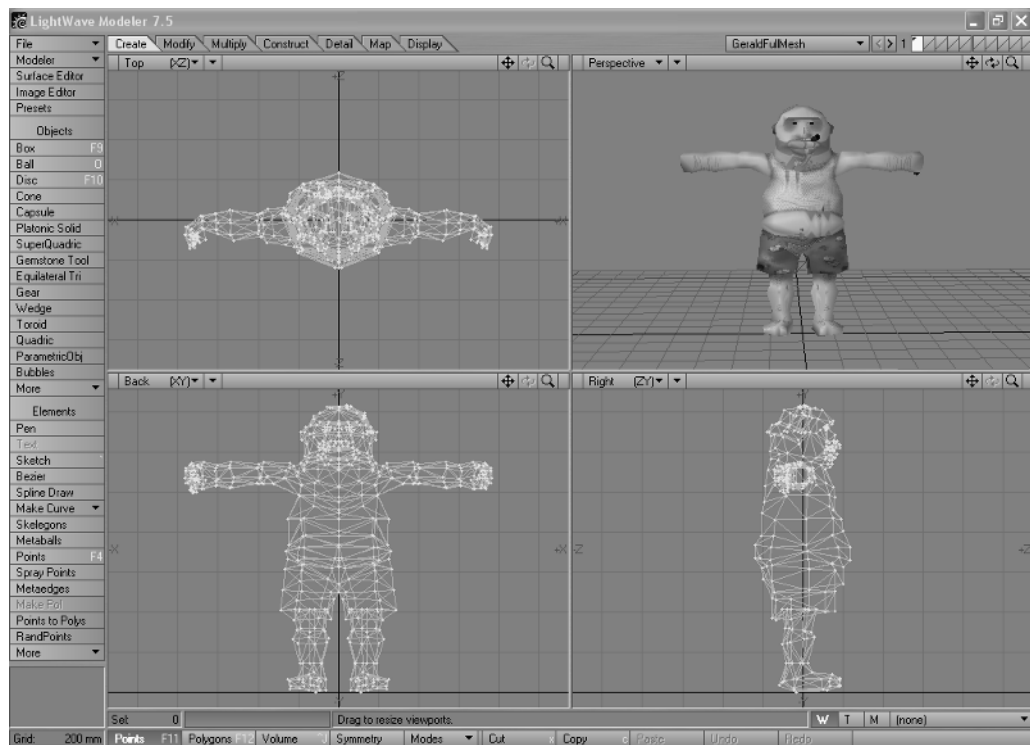


Figure 16.4 *The modeler interface in Lightwave 7.5*

and another for the body. When creating a head I prefer to build the polygons from scratch but when creating a body I like to start with a basic mesh. At this point all generality falls over and we look at how to build this geometry with Lightwave 3D. At the time of writing the latest Lightwave version is 7.5. For basic low-polygon mesh creation all the five versions of Lightwave are just as useful. Lightwave's user interface is shown in Figure 16.4.

This is typical of a modeler, splitting the screen into four user views. Three of the views are orthographic showing no perspective foreshortening. The interface is very configurable but by default the top left view shows a view looking down the 'y'-axis from above. Left and right in this view is the 'x'-axis and up and down represents the 'z'-axis. Bottom left is the front view looking straight down the z-axis. In this view left and right are again the x-axis but up and down is the y-axis. Bottom right shows the side view with left and right representing the z-axis and up and down again representing the y-axis. The final view in the top left is a perspective view shown using 'OpenGL' rendering, fast full color view. Clicking on the rotate icon for this view can rotate this view and the position can be moved using the transform icon. At any time pressing the 'g' key in an orthographic view centers the view over the position of the mouse. The less than '<' and greater than '>' keys zoom in and out. Pressing the 'a' key scales the views so that the model is fully visible in all the views.

When making a model you will use both point selection and polygon selection modes, which are activated by pressing the appropriate button at the bottom left of the screen. In point selection

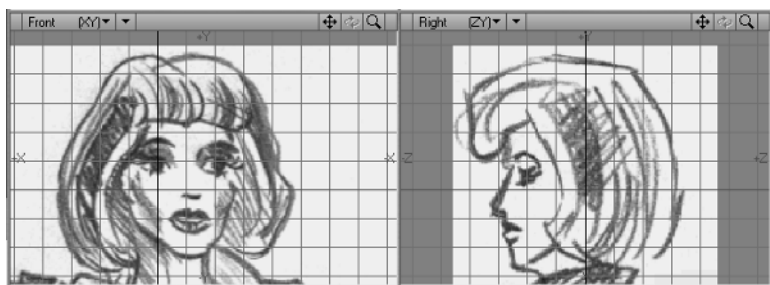


Figure 16.5 *Using a backdrop image in Lightwave 3D*

mode clicking on any point in the views will select that point; if you keep the left button down then you can select several points. Having released the left mouse button, clicking on any selected point deselects it. To add to the selection press the shift key while clicking on a point. To clear a selection click in the gray area to the bottom left of the screen. Polygon selection is done in much the same way but instead of clicking on a point you click on an edge and any polygons that share this edge are selected. By releasing the left mouse key and then clicking on selected polygons in one of the three views, you can easily select just the polygons that you want to work on. If you have a selection then you can zoom in to this by pressing the 'a' key while pressing the shift key.

We will start by loading a backdrop image. If you are working along with this tutorial then you will find a side and back view for Charlie in the 'Chapter16\Charlie\Images' folder. These images are called 'FrontSketch.tga' and 'SideSketch.tga'. To load a backdrop image press 'd' to open the display options dialog box. Select the 'Backdrop' tab. The front view is number 4 so select '4' and from the image drop-down choose 'Load image'. Navigate to the appropriate file and click 'open'. Repeat for view 4 but this time choose the side view. If you have selected correctly then you should have a view like Figure 16.5. To view the face close up, use a combination of the 'g' key to center the view over the mouse cursor and the '>' key to zoom in.

This rather simple sketch will help when creating the geometry for the face. The first step is to create some polygons for the mouth. To do this, select the point tool. This tool is available under the 'Create' tab at the top of the screen. Having selected Create, the buttons on the left will include 'Point' near the bottom left. Select this tool. Using the point tool you can create points by clicking with the right mouse key. Click around the outside of the lips and again around the inside. Because we are aiming at a symmetrical face, we can just build one side of the face and having accomplished the desired look we can mirror and weld the points where both sides join to create the final result. Figure 16.6 shows the first few polygons in the construction of the face.

To arrive at this result I first created the points that form the polygons around the lips. I then selected four points in a clockwise direction. Then, using the make polygon tool, which can be found under the 'Create' tab named 'Make Pol', I turned this selection of points into a single polygon. I repeated this for all the points in the lips. Then choosing 'Polygon Selection', all these polygons were selected, having just been created. If they are not selected, then they can be selected by drawing a ring around them, while pressing the right mouse button. Now pressing the 'q' key brings up a basic surface dialog. Here you can give a set of polygons a surface name. Type in 'Lips', set the surface color to red, set specularity to 60% and click the 'Smooth' button. This will ensure

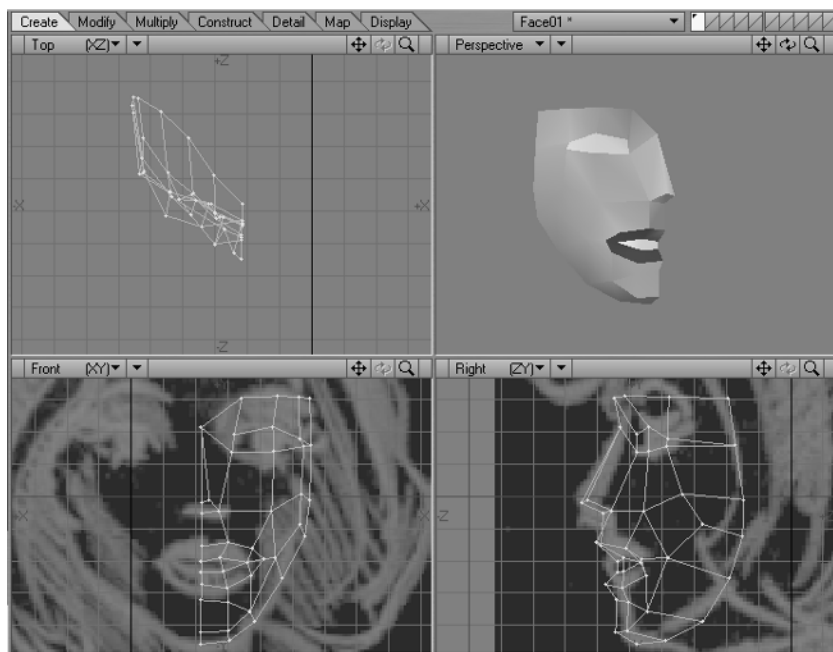


Figure 16.6 *Early work on Charlie's face*

that the lips appear smoothed rather than faceted. At this stage the geometry for the lips will be correct in the front view only. In the side view it will be totally flat. Continue to build polygons until you arrive at the model in Figure 16.6 in the front view. Now it is time to add depth to this geometry. To do this select points using the point selection tool and then press the 't' key, which activates the move tool. This tool allows you to move the selection, whether the selection is a point set or a polygon set. It is very easy to get confused about which point in the front view represents the same point in the side view. If in doubt waggle the point about and check to see what is moving in the side view. Remember that to deselect a set of points choose the point selection tool and then click in the gray area to the bottom left of the screen. Hopefully you have been able to create this geometry and color the polygons using the 'Surface' dialog box. But, if you feel like cheating, load 'Chapter16\Charlie\Objects\Face01.lwo', which is the model that you can see in the screen grab. It is now time to mirror this geometry and weld the points around the axis. The mirror tool is activated using 'Shift-V' or by choosing the 'Multiply' tab and selecting the 'Mirror' button. Click to define the axis and drag the mouse. A mirrored copy of the geometry is created. Deselect the tool and choose 'Point Selection'; click on a point around the join. If this point is indicated as a single point, then the mirror tool has successfully welded the points together. If you have any points that are not welded then weld them together now using the weld tool, which can be found under the 'Detail' tab. An alternative to welding each pair of points in turn is to use the 'Multiweld.ls' Lscript plugin, which is available on the CD under 'Utilities'. To use this, select a point on one side of the join and a point on the other using the point selection tool. Having selected several pairs of points in this way select the 'Construct' tab. Near the bottom left of the screen choose the Lscript

drop-down button and select 'LW_Lscript'; from the file open dialog box choose Multiweld.ls. This simple plugin joins pairs of points and welds them into a single point. At this stage you should have the face shown in Figure 16.7.

Creating Charlie's body

Now it is time to start on Charlie's body. The Lightwave interface uses a layer style, in much the same way as a Photoshop or a Paint Shop Pro bitmap file can use several layers in the creation of the final image. Having created the face in layer 1, we will now move to layer 2 to start work on the body. The layer palette is in the top right corner of the screen. Here you will see 10 rectangles each cut through by a diagonal. If you click in the lower part then you set this as the active background.

A background in Lightwave shows through as a black line image in all the views. It is useful for aligning your geometry to existing models. This is precisely what we wish to do now. So click on the upper part of layer 2 and the lower part of layer 1. You should be able to see the face as a black line in all the views. If you have been working through the tutorial then you will have the backdrop images as a useful guide. Use the 'g' and '<' keys to zoom out and recenter the views so that all of the guide backdrops are visible.

The first step is to create the waist. We will use the disc tool. This is accessed from the 'Create' tab. Click on the 'Disc' button then drag an ellipse in the 'Top' view. The ellipse will define the size of Charlie's waist. Before leaving this tool, press the 'n' key. This brings up a numeric dialog box. This allows you to set various parameters for the disc tool. Many of Lightwave's tools have numeric options, which are all accessed via the n key. In the option for the number of sides select '16'. Drag again in the 'Front' view and you will see an elliptical cylinder appear. This mesh has two polygons with 16 vertices. These two polygons are likely to cause havoc with any real-time engines, so it is best to remove them. The problem polygons are the top and bottom faces of the cylinder. Draw a ring around one of them using the right mouse button in 'Polygon Selection' mode and when the polygon goes yellow you can delete it by pressing the 'Delete' key. An alternative way to select polygons with more than four sides is to select the 'Polygon Statistics' dialog box by pressing the 'w' key. Using this dialog box you can quickly find geometry that may cause problems with the real-time engine. The dialog box contains buttons to select '+' or deselect '-' polygons with one, two, three, four or more than four sides. If you are working through this tutorial then this dialog box should indicate one polygon with more than four sides. Press the + button next to this and the offending polygon should be highlighted in yellow. Again to delete the polygon press the Delete key. You should now have a cylinder with no top or bottom.

To add geometry we will knife through the upper part of the cylinder in the 'Front' view. Select the 'Knife Tool' under the 'Construct' tab. Draw a line in the front view by clicking on the Front view and then dragging the handles of the line. Press the space bar to confirm the knife action.

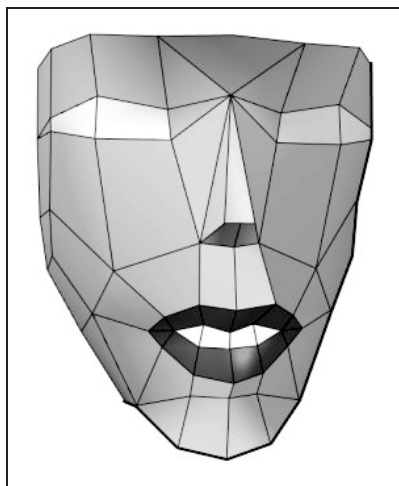


Figure 16.7 Stage 2 in creating Charlie's face

You should now have an additional row of points in the elliptical cylinder. Because Charlie's body is symmetrical we are going to delete the left half and concentrate on the right half for the construction. Once we have got near to completing the right half we will mirror it in just the same way we did with the face, welding the points along the mirroring axis. To delete the left half simply draw a ring around the polygons on the left in 'Polygon Selection' mode and press the 'Delete' key. The highlighted polygons will be deleted. Now select the upper row of points by switching to 'Point Selection' mode and drawing a ring around these points using the right mouse button. Press the 't' key to activate the move tool and drag these up until they are at shoulder height. Now knife through the geometry just below the bust line. Highlight this latest row of points and use the stretch tool, which is accessed using the 'Modify' tab or by pressing the 'h' key. The stretch tool allows you to move a group of points or polygons. The first click defines the center of the stretch. Dragging the mouse then enlarges or shrinks the selection in the axis of the drag. You want to stretch this row of points until the size of this section is comparable to the sketch. It is now that you will realize the usefulness of this simple sketch. Without it, adding geometry requires a very good eye and lots of experience. Repeat the knifing and stretching until you have four new rows of points. Take a look at the object called 'Body01.lwo' in the 'Objects' folder for Charlie to see where you need to have arrived at.

It is now time to cap the neck area. To do this highlight the upper row of polygons and press the '=' key. This hides any geometry that is not selected. This does not mean it is deleted; it is simply hidden from view. Pressing the '\ ' key at any time will restore the full geometry. To cap the neck area we will make some polygons out of sets of four vertices in the top row of points. The outer polygon will have just three vertices. By selecting the newly created polygons and choosing the smooth shift tool we can extrude this set of polygons and create a new row. The smooth shift tool is accessed from the 'Multiply' tab. The selection of polygons will be shifted along the vertex normals as you drag left and right. The neck has just 12 vertices in a row so we must weld some points together to achieve this. Take a look at 'Body02.lwo' to see the stage we have arrived at.

To continue we need to smooth shift polygons near the armpits to create the geometry that will form the arms. This will be knifed to create an indent for the elbow. Knife through the lower section and form the hips, in the same way that we have created the upper body. The lowest row of points will be used to form an area in which to create the legs. Draw a ring around this lowest row of points and press '=' to show just this selection. In the 'Top' view select just the front and back points for the points nearest the middle and weld these together. Repeat this for the next points. Then using the drag tool reshape the remaining points so that they form a hexagon. Select these points and choose 'Make Pol' to create a polygon from these points. Press the '\ ' key to show again the full geometry and select just the newly created polygons. Smooth shift allows you to create the beginnings of a leg. The knife and stretch tools will allow you to shape this leg close to the geometry of the backdrop images. To get closer to the final geometry you will need to use a combination of the drag tool and the move tool with point selections to drag and move the points into their final locations. This part of the process is all about judging which points are which in the different user views. Lightwave 7.5 allows you to select points in the perspective view and this can certainly make a difficult-to-locate point much easier to find. 'Body03.lwo' is the stage we have now reached.

To add Charlie's collar, use the point selection tool to select the points where the base of the collar will be formed. Copy this point selection by pressing the 'c' key. You can then paste this

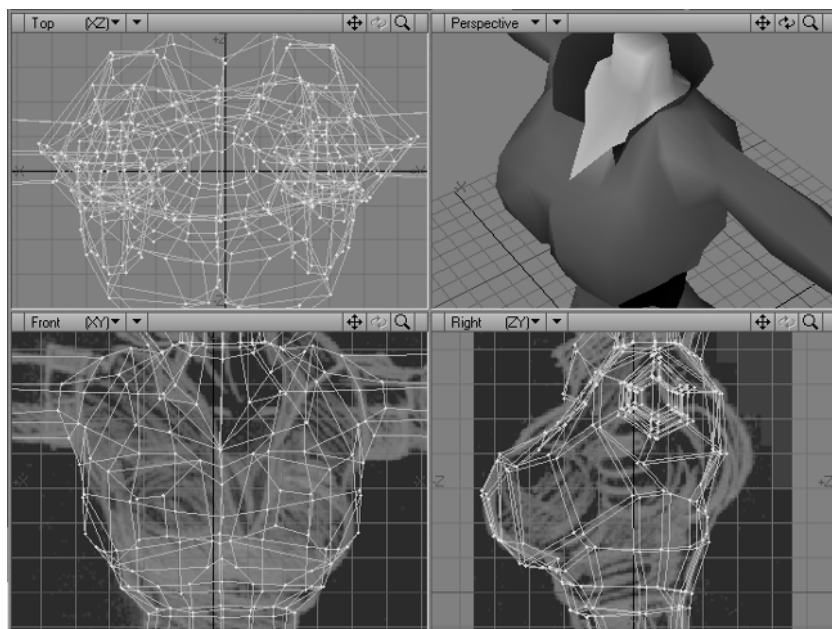


Figure 16.8 *Creating Charlie's upper body*

to another layer. Move these new points to where the top of the collar will be. Then press **c** again to copy the new points. Move back to the original layer and paste the points. Now use the point selection tool to select the base points and the new points. Press the '=' key so that only the selection is visible and select sets of four points and use 'Make Pol' to turn the point selection into a polygon. If everything went correctly then Charlie should now be the proud owner of a new collar.

Creating Charlie's hands and feet

To form Charlie's hand we start with the last two polygons of her left arm and select these. Using the smooth shift tool extend these polygons out to form the area where Charlie's palm will be. We need three polygons to create the thumb, index finger and remaining fingers. To get these three polygons repeat the smooth shift. Use one side polygon and the two end polygons to form the digits. Because the smooth shift tool moves a set of polygons together we will use the bevel tool. This tool deals with each polygon in isolation. In addition to being able to drag the mouse left and right to move the new geometry along the vertex normals, the bevel tool allows you to enlarge or shrink the end polygon. Use this feature to shape the fingers. You should by now have a roughly shaped hand. We are dealing with a low-polygon character so we cannot put very much detail into a hand, but it is important that the shape is at least a rough representation of a hand. In much the same way we arrived at the final geometry for the body, use the drag and move tools with point selections to shape the hand. Having created the left hand use the polygon selection tool to highlight the left hand and then use the mirror tool to duplicate a mirrored image of this polygon selection. Now delete the end polygons from Charlie's right arm and weld the free-floating right hand to the end of her arm.

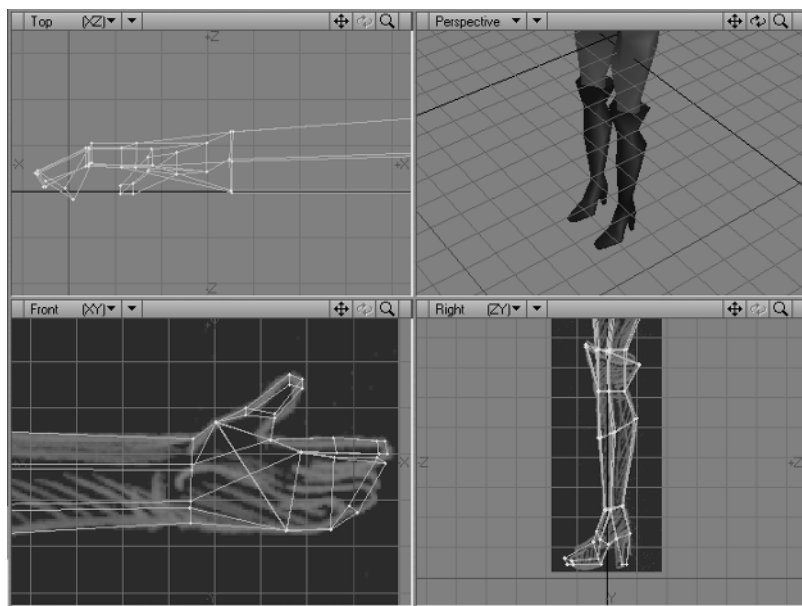


Figure 16.9 *Creating low-polygon feet and hands*

To create the feet, we first use smooth shift on the end polygons of the left leg to form the left ankle. In the same manner as the hands, we will create just a single foot. Having shaped the foot to our satisfaction, we highlight this group of polygons and mirror them to create the right foot, welding the right foot onto the polygons at the end of the right leg. Then change the selection to the front face of these new polygons. Use smooth shift again; this time the polygons will be created pointing forward rather than down. Select the back lowest polygon and smooth shift this to form the heel. Knife through the forward-facing polygons twice. You now have all the polygons necessary to shape a foot. The final shaping is done using a combination of dragging and moving of point selections. The last task in creating Charlie's body is to add the top of her boots. This is done in the same way we created her collar. Because we need to create two sets of these polygons, one for the left boot and one for the right, it is best to create a single set, mirror it and weld the new one in place.

Finishing Charlie

The final stage of creating Charlie is to weld her head onto her body and shape her hair. Copy Charlie's head to the body layer and weld the points in her face to the neck. To form Charlie's hair, highlight the points in the left edge of Charlie's face and copy these to a new layer. With Charlie showing as a background layer rotate the newly pasted points to the back of Charlie's head. Copy the points back to Charlie's main layer. Create polygons from the new points and the edge of Charlie's face. Knife through the new polygons twice. Select the points created by this knifing and use the move tool in the 'Top' view to make Charlie's head more rounded. Select 'Mirror' and weld the polygons that form the side of Charlie's head to create the other side. Weld the back seam

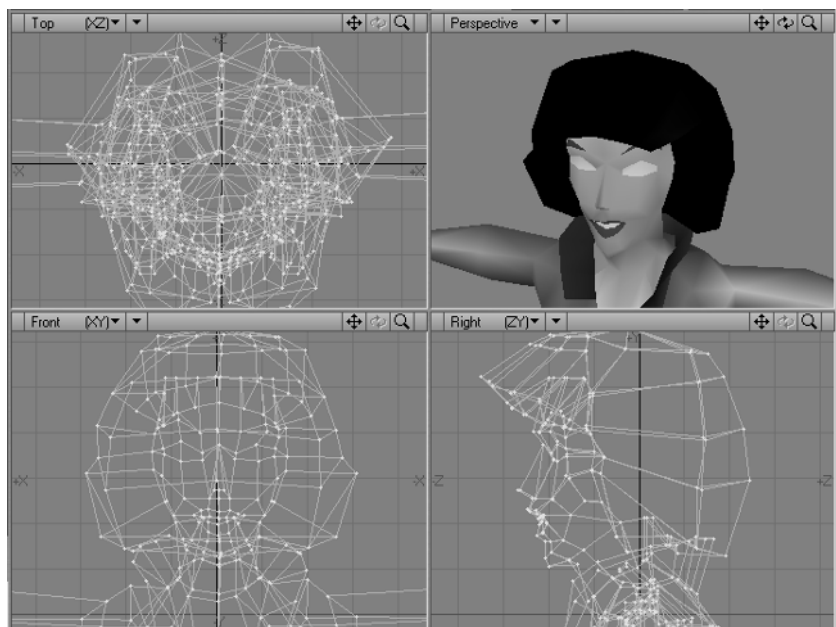


Figure 16.10 *Joining Charlie's head to the body*

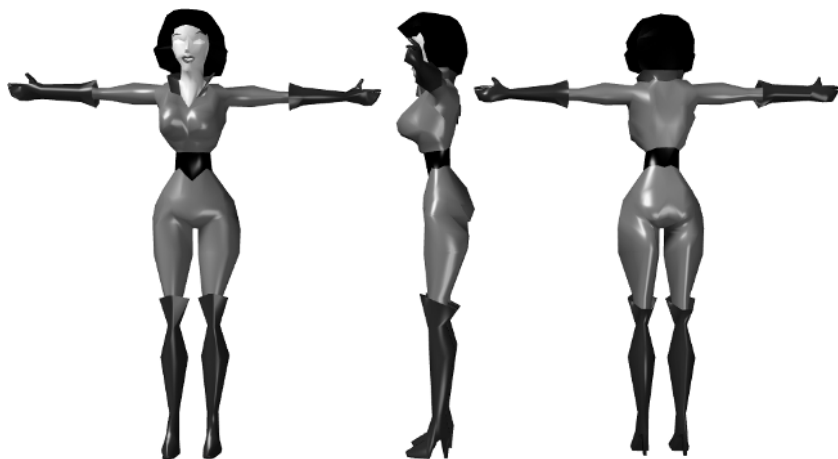


Figure 16.11 *The Charlie model finished without textures*

and weld these new polygons onto Charlie's neck. Select the top row of points. Press the '=' key to hide all the other geometry. Use the point tool to create points for the top of Charlie's head. Highlight sets of four points and use 'Make Pol' to turn the selection into a polygon. Press the '\ ' key to show the full geometry again. Now highlight the polygons that will form Charlie's hair. Use the smooth shift tool to move this polygon set out. It only remains to use the drag and move tools to create the final shape for the geometry.

If everything has worked out then you should have the geometry featured in Figure 16.11.

Creating a more cartoon-style character

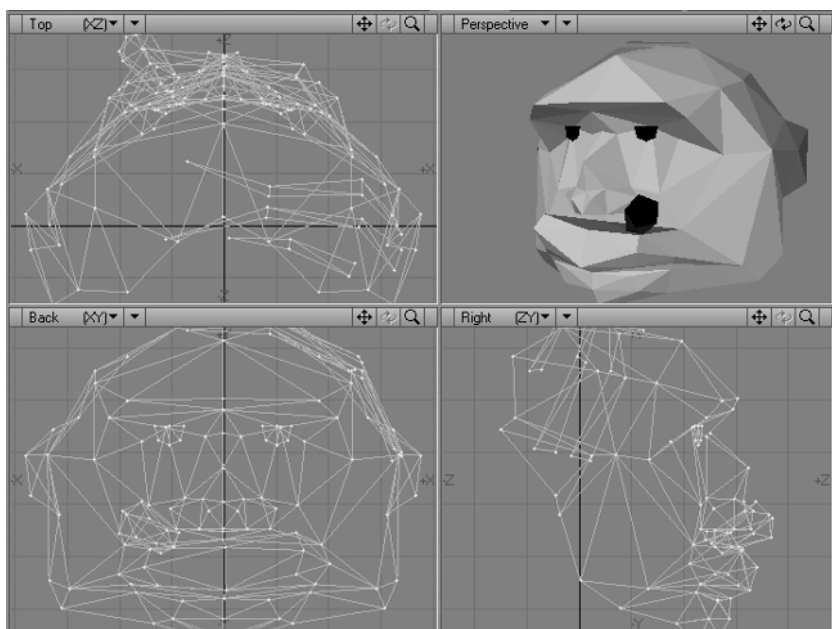


Figure 16.12 *Creating Gerald's head*

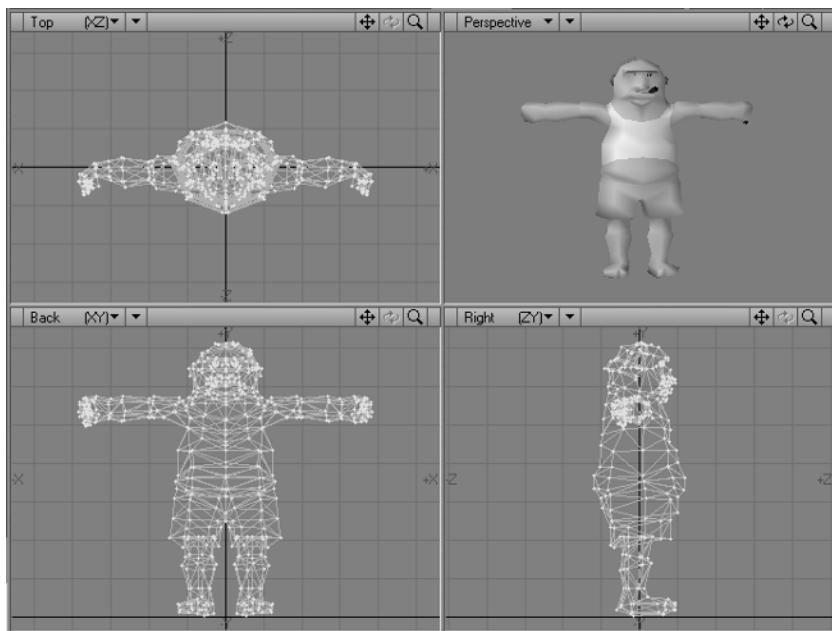


Figure 16.13 *Gerald's full mesh*

Although Charlie is a caricature she is quite life like. You can use CGI to create very cartoony type characters. The development of a character is a complex and highly personal exercise. I highly recommend sketching before starting any modeling. Feeling for form with a pencil is a much faster process than creating geometry.

Figure 16.13 shows the mesh 'Examples/Chapter16/Gerald/Objects/Gerald.lwo'. With low-polygon meshes the artist has to make a compromise between the geometry they would like to use to create their characters and the geometry that current computers can transform and render at 25 frames per second or faster. At the time of writing, games consoles and graphics cards are allowing 50 K poly scenes where previously we had 5 K scenes. But even with 50 K polys there is still a huge compromise over TV- or film-rendered CGI. Here 500 K scenes are usual and 1 million plus poly scenes not that unusual. But we still want our characters to look great. We can create the illusion of much more geometry by the intelligent use of textures on our characters. Figure 16.14 shows the Gerald mesh with textures added. 'Examples/Chapter16/Gerald/Gerald.htm' allows you see this textured mesh as a turn around. Gerald in full color looks so much better than the simple smooth shaded mesh. The next step is to introduce animation into the character.



Figure 16.14 *Gerald with textures*

Creating a skeleton

In order to animate a character mesh we need a way to move certain points in the mesh one way and others in a different manner. In other words we need a skeleton. When the bone that represents the left upper arm moves, then all the points in that left arm must move, while the rest of the body must remain unaffected. All CGI packages have some way to handle this. Lightwave is no exception. To create the skeleton in the first place we use the modeler application, creating a specialized polygon, which Lightwave calls a skelegon. A skelegon is simply a two-pointed polygon that is assigned not to display when we render the geometry. You can create skelegons in Lightwave using the 'Create>Skelegons' button; click on the 'Create' tab at the top, the 'Skelegons' button is about half way down on the left in the default configuration. Now you can draw skelegons in the view ports. You can also move and drag the vertices about and weld skelegons to other parts. When creating skelegons for a scene that will be exported for Director use,

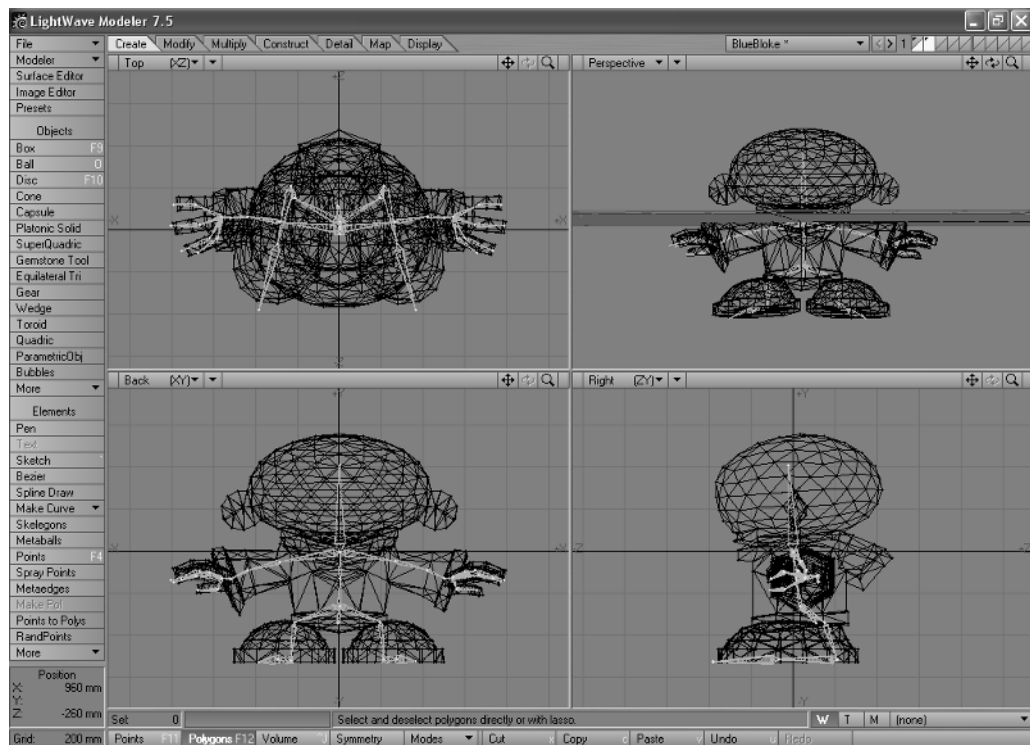


Figure 16.15 *Creating a skeleton in Lightwave 3D*

keep in mind that all skelegons must be connected to a root bone. Once you have created the skeleton, highlight each bone and name it.

Figure 16.16 shows the full hierarchy for the skeleton used for the sample 'Examples/Chapter16/XBlokes/XBlokes.dir'. Notice how the bones are connected: the 'Waist' is connected to the 'Root', the 'Torso' to the 'Waist' and so on. Using this hierarchy, as we move or rotate a parent the descendants of that bone will also be moved or rotated. This means that as the upper arm rotates so the lower arm rotates, but the lower arm can have its own rotation built on top of the parent's movement. For example, the upper arm can rotate to point out in front of the body while the lower arm bends at the elbow. Once you have created a skeleton, the software needs some way of connecting a bone to a set of vertices. Lightwave uses weight maps for this purpose.

Assigning weight maps

Figure 16.17 shows a selection of vertices; to create a weight map click on the 'W' button in the lower right of the modeler application. In the drop-down to the right of the buttons select 'new'. Type the name of the weight map and retain the defaults in the panel that is shown. It is useful to keep the same name for both the weight map and the bone.

Figure 16.18 shows the full list of weight maps used in the example.

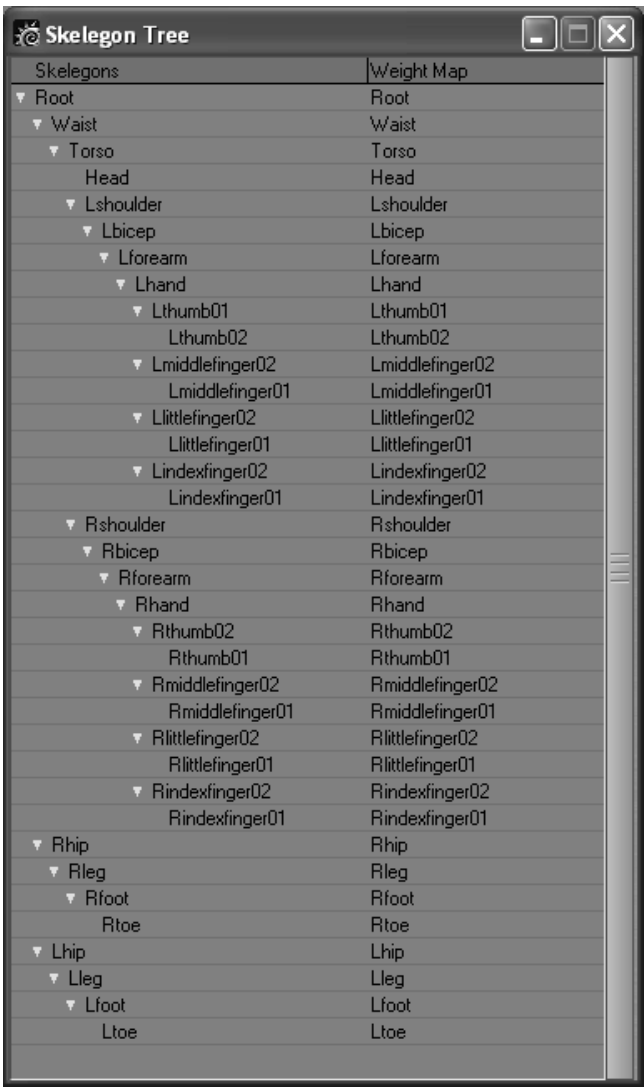


Figure 16.16 *Displaying the hierarchy and the weight map for each bone*

Animating the bones

It is very important that every vertex in the mesh is assigned to at least one weight map otherwise the mesh will not export correctly. Now you can load the object into the ‘Layout’ application; Lightwave splits the modeling and animating into two different applications. Having loaded the mesh, select the ‘Items’ tab at the top and then choose ‘Add>Bones>Convert Skelegons into Bones’. At this stage all the specialized polygons become bones that you can animate. If everything went well, rotating the torso should result in the upper body moving while the lower body remains stationary.

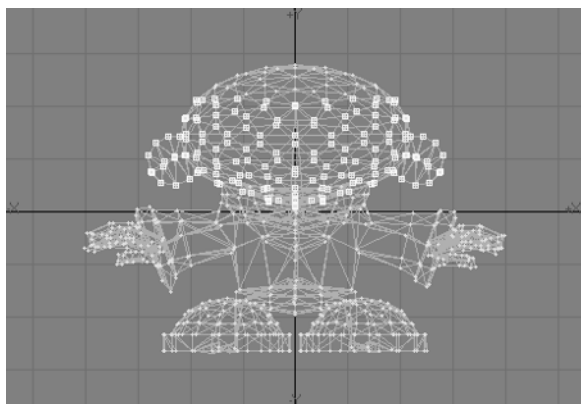


Figure 16.17 *Selecting a set of points*

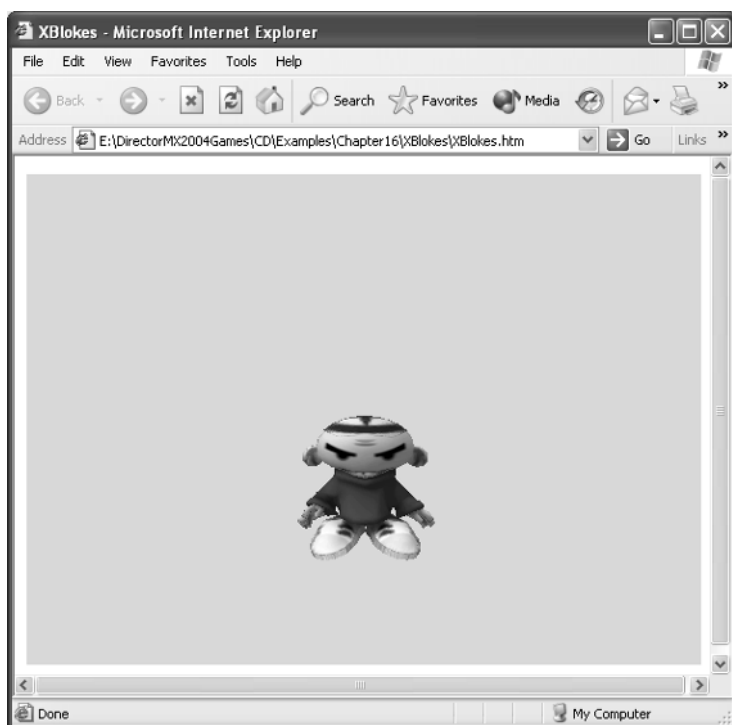


Figure 16.19 *'Examples/Chapter16/XBlokes/XBlokes.htm' in Internet Explorer*

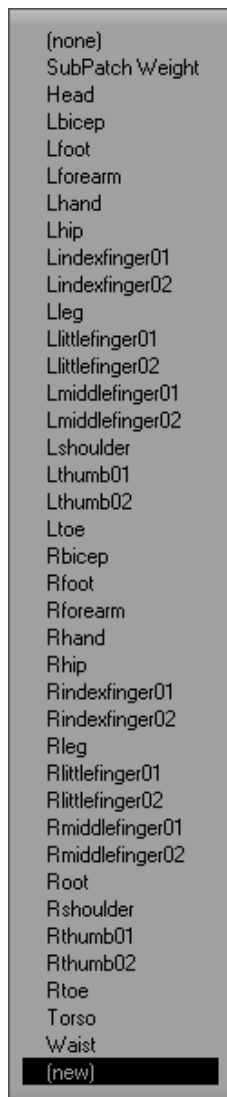


Figure 16.18 *A list of weight maps*

Exporting to the w3d format

You can now animate the character. Having completed the animation, select 'File>Export> Shockwave3D_Export' to export the scene in a format that you can use in Director.

Figure 16.19 shows the sample 'Examples/Chapter16/XBlokes/XBlokes.htm' as it appears when published in Director and viewed in Internet Explorer.

Summary

Most readers who will come at real-time 3D from a coder's perspective will find the content of this chapter provides an insight into the skills of the artists who create the meshes that their code transforms and renders. Hopefully you found the techniques interesting and are encouraged to create your own meshes. The techniques described, although exclusively from a Lightwave perspective, can be applied to other modeling packages that focus on polygonal modeling.

17 3D basics

Starting with version 8.5 a 3D Xtra was added to Director, developed by Intel. The 3D Xtra also adds a great many additional scripting commands. Using this Xtra it has been possible to create some extremely dynamic games that are suitable for viewing within a browser or inside a platform-dependent executable. For the game developer 3D games represent a significant increase in complexity. For the game player they add an immersive depth that is lacking in the 2D alternatives. In this chapter you will be introduced to the basics of creating a 3D world on a 2D screen. The presentation is general and is intended to take the mystery out of what is happening under the hood when you begin to develop 3D games using Director MX 2004. 3D games are the most popular type of game, now that low-cost hardware is capable of displaying them at reasonable frame rates and with Director MX 2004 you have most of the developer tools available to make the next blockbuster.

Describing 3D space

First let's imagine a small box lying on the floor of a simple room (see Figure 17.1).

How can we create a dataset that describes the position of the box? One method is to use a tape measure to find out the distance of the box from each wall. But which wall? We need to have a frame of reference to work from.

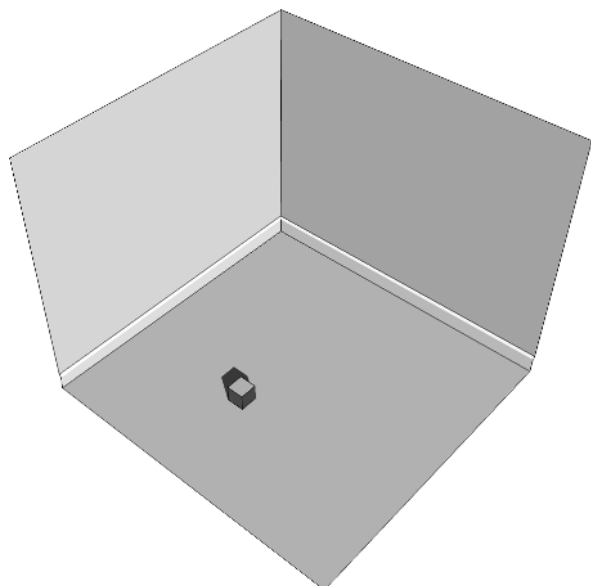


Figure 17.1 *A simplified room showing a small box*

Figure 17.2 shows the same room only this time there are three perpendicular axes overlaid on the picture. The point where the three axes meet is called the *origin*. The use of these three axes allows you as a programmer to specify any position in the room using three numeric values. In Figure 17.2 the two marked lines perpendicular to the axes give an indication of the scale we intend to use. Each slash on these lines represents 10 cm. Counting the slashes gives the brick as 6 along the 'x'-axis and 8 along the 'z'-axis. The box is lying on the floor so the value along the 'y'-axis is 0. To define the position of the box with respect to the frame of reference we use a

vector: [6, 0, 8]. The direction of the axes is the scheme used in the Director 3D Xtra. The y-axis points up, the x-axis points right and the z-axis points out of the screen.

Transforming the box

To move the box around the room we can create a vector that gives the distance in the 'x', 'y' and 'z' directions that you intend to move the box. For example, if we want to move the box 60 cm to the right, 30 cm up and 20 cm towards the back wall, then we can use the vector [6, 3, 2] (recall that the scale for each dash is 10 cm) to move the box.

The sum of two vectors is the sum of the components:

$$[x, y, z] = [x1, y1, z1] + [x2, y2, z2]$$

where

$$x = x1 + x2, y = y1 + y2 \text{ and } z = z1 + z2$$

For example,

$$[12, 3, 10] = [6, 0, 8] + [6, 3, 2].$$

Fortunately Lingo and JavaScript commands in Director MX 2004 allow you to transform a model by using the following code, found in 'Examples/Chapter17/translate.dir' and 'Examples/Chapter17/translateLingo.dir'.

```
//JavaScript
1  var pModel;
2
3  function beginSprite(){
4      member("axes").resetWorld();
5      pModel = member("axes").getProp("model", 1);
6  }
7
8  function enterFrame(){
9      pModel.translate(0, 0, 2);
10 }
```

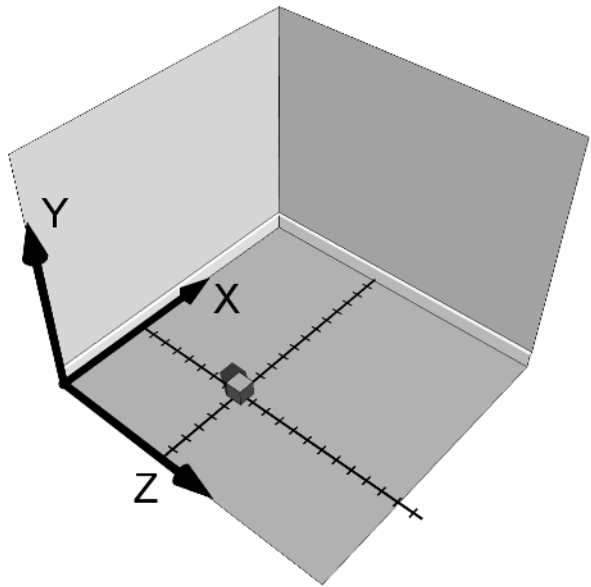


Figure 17.2 A simplified room with overlaid axes

```
--Lingo
1  property pModel
2
3  on beginSprite
4    member("axes").resetWorld()
5    pModel = member("axes").model[1]
6  end
7
8  on enterFrame
9    pModel.translate(0, 0, 2)
10 end
```

Listing 17.1

Describing an object

The simplest shape that has some volume has just four points or *vertices*. A tetrahedron is a pyramid with a triangular base. We can extend the idea of a point in 3D space to define the four vertices needed to describe a tetrahedron. Before we can draw an object we also need to define how to join the vertices. This leads to two lists: a list of vertices and a list of faces or *polygons*.

The vertices used are

```
A: [ 0.0, 1.7, 0.0]
B: [-1.0, 0.0, 0.6]
C: [ 0.0, 0.0, -1.1]
D: [ 1.0, 0.0, 0.6]
```

To describe the faces we give a list of the vertices that the face shares.

```
1: A, B, D
2: A, D, C
3: A, C, B
4: B, C, D
```

Although the triangles ABD and ADB appear to be the same, the order of the vertices is clearly different. This ordering is used by many computer graphics applications to determine whether a face is pointing towards the viewer or away from the viewer. Some schemes use points described in a clockwise direction to indicate that this face is pointing towards the viewer. Other schemes choose counter-clockwise to indicate forward facing polygons. In this book we have used counter-clockwise; there are no advantages or disadvantages to either scheme, it is simply necessary to be consistent. Think about the triangle ABD as the tetrahedron rotates

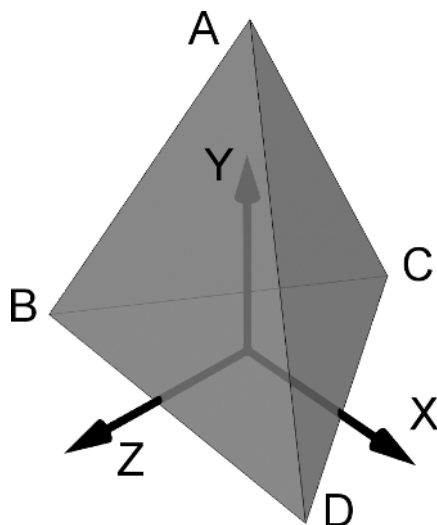


Figure 17.3 A tetrahedron

about the 'y'-axis. If this rotation is clockwise when viewed from above then the vertex B moves right and the vertex D moves left. At a certain stage the line BD is vertical. If the rotation continues then B is to the right of D. At this stage in the rotation the face ABD is pointing away from the viewer. Because we know that the order of the vertices read in a counter-clockwise direction should be ABD, when the order changes to ADB, the triangle has turned away from the viewer. This is very useful because in most situations it is possible to effectively disregard this polygon (if an object is transparent then it will be necessary to continue to render back-facing polygons). Vertex order is always the most efficient way to compute polygons that do not need to be painted.

Polygon normals

See Figure 17.4. A normal is simply a vector that points directly out from a polygon. It is used in computer graphics for determining lighting levels amongst other things. Most developers will use an external 3D modeling and animation tool to create the geometry and animations that will make up the 3D world. Once created these are exported into a format that Director can use. The format has the extension w3d. Most exporters will export the normals along with the vertex position details and lists of polygon data that point to a set of vertices and normals.

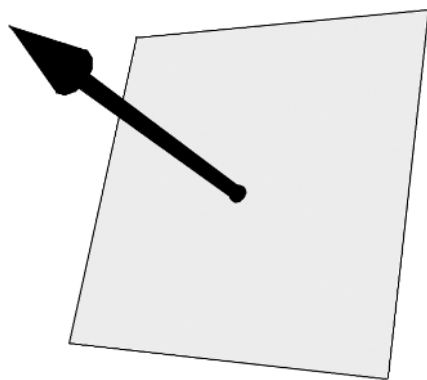


Figure 17.4 *A polygon normal*

Rotating the box

When using Director you will want to be able to rotate an object. Rotations in 3D space are not as clear cut as positions or scales. An object when placed at a particular location will always go to that position. However, rotation is a much more exacting science and as you start to play with 3D space you will realize that creating rotations that are repeatable is more difficult than it may at first appear. You can rotate an object in Director using either Euler angles or angle and axis rotation.

Euler angles

See Figure 17.5. We store an angle for the heading, the pitch and the bank. The heading is the rotation about the 'y'-axis, the pitch is the rotation about the 'x'-axis and the bank is the rotation about the 'z'-axis. Using three numbers we can describe the orientation. The numbers used however are interdependent. The order in which they are specified is important. If one angle rotates an object through 90° then one axis is mapped to another and one axis of rotation is lost. In 3D animation programs this is described as gimbal lock.

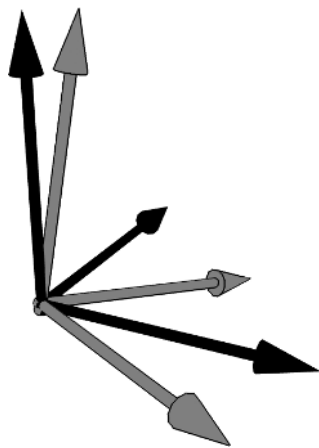


Figure 17.5 *Euler angle rotation*

```
//JavaScript
1  var pModel;
2
3  function beginSprite(){
4      member("axes").resetWorld();
5      pModel = member("axes").getProp("model", 1);
6  }
7
8  function enterFrame(){
9      pModel.rotate(0, 2, 0);
10 }

--Lingo
1  property pModel
2
3  on beginSprite
4      member("axes").resetWorld()
5      pModel = member("axes").model[1]
6  end
7
8  on enterFrame
9      pModel.rotate(0, 2, 0)
10 end
```

Listing 17.2

The above code can be found in 'Examples/Chapter17/rotate.dir' and 'Examples/Chapter17/rotateLingo.dir'.

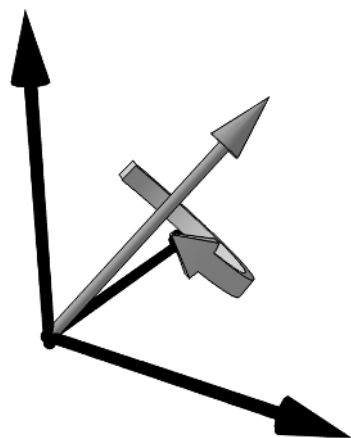


Figure 17.6 Angle and axis rotation

Angle and axis rotation

See Figure 17.6. The values used are an angle, θ and a vector, $A = [x, y, z]^T$, which represents the axis of rotation. Although this method avoids gimbal lock it is much more difficult to interpolate. The angle axis method can be expressed in Director using the following code.

```
//Javascript
1  var pModel;
2
3  function beginSprite(){
4      member("axes").resetWorld();
5      pModel = member("axes").getProp ("model", 1);
6  }
7
8  function enterFrame(){
9      pModel.rotate(pModel.worldPosition, vector(0, 1, 0), 2);
10 }
```

```
--Lingo
1  property pModel
2
3  on beginSprite
4    member("axes").resetWorld()
5    pModel = member("axes").model[1]
6  end
7
8  on enterFrame
9    pModel.rotate(pModel.worldPosition, vector(0, 1, 0), 2);
10 end
```

Listing 17.3

The code in Listing 17.3 can be found in 'Examples/Chapter17/rotateAA.dir' and 'Examples/Chapter17/rotateAALingo.dir'. Notice the new call to 'rotate' at line 9 in both the Lingo and JavaScript versions. The first parameter is a world position, the second parameter is a vector that represents the axis of the rotation, in this case around the 'y'-axis because we have a vector that is simply pointing directly up the 'y'-axis and the third parameter is the number of degrees that the rotation moves around this chosen vector.

Rotation about a point other than the origin

The syntax used in Listing 17.3 allows the programmer to rotate an object around an arbitrary point in space. By default the rotation occurs around the origin of the object; the point in the object where 'x', 'y' and 'z' are all equal to zero. If we want to rotate a box about its top, left, front corner then we would use the code in Listing 17.4.

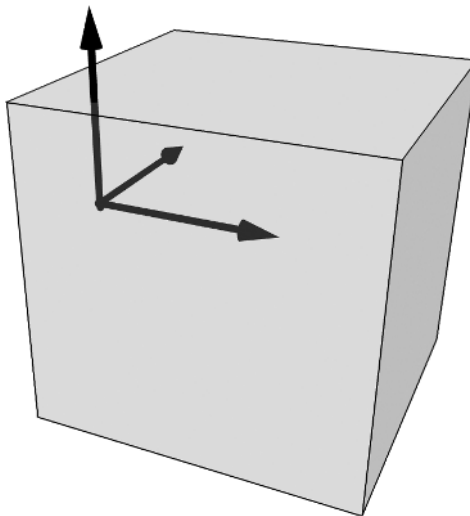


Figure 17.7 *Rotation about a pivot point*

```
//Javascript
1  var pModel, pOffset;
2
3  function beginSprite(){
4      member("axes").resetWorld();
5      pModel = member("axes").getProp("model", 1);
6      pOffset = vector(200, 0, 0);
7  }
8
9  function enterFrame(){
10     pModel.rotate(pOffset, vector(0, 1, 0), 2);
11 }
```

```
--Lingo
1  property pModel, pOffset
2
3  on beginSprite
4      member("axes").resetWorld()
5      pModel = member("axes").model[1]
6      pOffset = vector(200, 0, 0);
7  end
8
9  on enterFrame
10     pModel.rotate(pOffset, vector(0, 1, 0), 2);
11 end
```

Listing 17.4

The code in Listing 17.4 can be found in 'Examples/Chapter17/rotateAAb.dir' and 'Examples/Chapter17/rotateAAbLingo.dir'.

Scaling the object

The size of the object has so far been unaffected by the operations considered. If we want to scale the object up or down we can use another scripting method.

```
//Javascript
1  var pModel, pCount, pUp;
2
3  function beginSprite(){
4      member("axes").resetWorld()
5      pModel = member("axes").getProp("model", 1);
6      pCount = 0;
7      pUp = false;
```



```

8  }
9
10 function enterFrame(){
11   if (pUp){
12     pModel.scale(1.02, 1.02, 1.02);
13     pCount++
14     if (pCount==20){
15       pCount = 0;
16       pUp = false;
17     }
18   }else{
19     pModel.scale(0.98, 0.98, 0.98);
20     pCount++;
21     if (pCount==20){
22       pCount = 0;
23       pUp = true;
24     }
25   }
26 }

```

--Lingo

```

1  property pModel, pCount, pUp
2
3  on beginSprite
4    member("axes").resetWorld()
5    pModel = member("axes").model[1]
6    pCount = 0
7    pUp = false
8  end
9
10 on enterFrame
11   if pUp then
12     pModel.scale(1.02, 1.02, 1.02)
13     pCount = pCount + 1
14     if pCount = 20 then
15       pCount = 0
16       pUp = false
17     end if
18   else
19     pModel.scale(0.98, 0.98, 0.98)
20     pCount = pCount + 1
21     if pCount = 20 then

```

```

22      pCount = 0
23      pUp = true
24  end if
25  end if
26 end

```

Listing 17.5

The code in Listing 17.5 can be found in ‘Examples/Chapter17/scale.dir’ and ‘Examples/Chapter17/scaleLingo.dir’.

‘z’-buffers

When your scene is painted, the Director Xtra works its way through all the triangles in your scene. First they are transformed, then they are converted from 3D into screen coordinates. The painting order of the triangles does not take into consideration distance from camera. So there needs to be a method of making sure that triangles close to camera block those further away. The ‘z’-buffer handles this problem in a simple and elegant way. Before the 3D Xtra paints a pixel it first checks in the z-buffer. A z-buffer is simply an array of data that matches the color data; for each screen point ‘(x, y)’ there is a corresponding point ‘(x, y)’ in the z-buffer. A screen point has color information stored while the z-buffer contains the distance from camera for each pixel. As the frame is painted the z-buffer is queried for each pixel of each triangle. If the distance for the pixel to be painted is less than that saved in the z-buffer then the current pixel must come in front of that already stored in the z-buffer. This simple technique handles all the complex clipping of triangles. But it does cause a problem when dealing with transparent pixels. Here the content of the z-buffer cannot be totally relied upon. In practice you will find that when a transparent pixel is drawn the z-buffer is updated to the distance to the transparent pixel rather than the opaque pixel that is behind it. As a consequence you will find that if you use lots of transparent triangles, particularly if a transparent pixel overlaps with another, then you will regularly get painting errors where the scene does not look as intended.

Hither and yon

An additional point to bear in mind concerns *hither* and *yon*. A camera has a clipping distance set for near to the camera (hither) and far away (yon). If you find that your triangles suddenly cut off then it is likely that the settings for hither or yon need to be adjusted. More information on camera properties is provided in the next chapter.

Summary

The basic operations presented in this chapter give you an overview of how a scene is constructed from vertices and triangles and then moved, rotated and scaled. Director MX 2004 provides all the tools to manipulate this 3D environment and in the next few chapters we will look at how you can

use these powerful additions to Director. Although it is possible to create great games using Director and the minimum of mathematics, you will find that the more confident your maths the easier you will find solutions to 3D world problems. You will find that if you are prepared to understand some basic trigonometry and how vectors and matrices work then there will be lots more you can do with your games. Appendix A gives a grounding in the maths you will need for your games. The next chapter looks in detail at how the data for your scene is actually stored in a way that Director can access and manipulate at real-time speeds.

18 What's in a w3d file?

In order to use 3D in Director you will need to learn about the contents of the only 3D file format that Director can import. These files have a w3d extension. Many 3D packages have an exporter that will create content in the w3d format. Some work better than others, but a feature of all the exporters is that to make sure the content looks the same in Director as it does in the 3D package requires experience and a knowledge of the limitations of both real-time 3D and Director in particular. In this chapter we explore the content of the file. A very useful Xtra that is essential when developing 3D games is 3D Property Inspector (3DPI) (see Figure 18.1). It is downloadable from <http://www.3dpi-director.com> and is highly recommended; without it you will find it very frustrating trying to discover what is inside your 3D worlds and how to access the various elements that are embedded within them.

Overview

To describe a 3D world to allow for real-time 3D rendering involves a huge dataset. The content of a w3d file breaks down into

- textures
- shaders
- model resources
- models
- motions
- lights
- cameras.

Each of these basic components is accessed using the same type of syntax.

```
--Lingo
member("3D World").texture[x] or
member("3D World").texture("textureName")
```

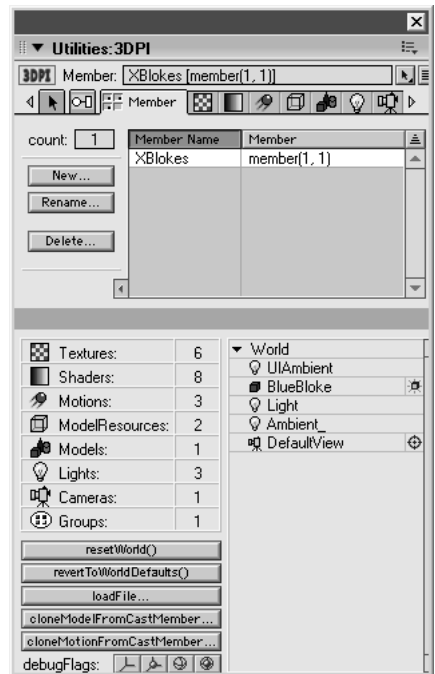


Figure 18.1 Using 3D Property Inspector

```
--JavaScript
member("3D World").getPropRef("texture", x)
```

Notice that JavaScript access requires the use of a function call, either ‘getPropRef’ to get a reference to an object or ‘getProp’ to get access to a value.

We will look at each of the basic properties in turn.

Textures

A texture is simply a bitmap file; the w3d format stores the bitmap in jpeg compressed format. Most exporters allow you to control the amount of compression that the textures experience when being converted. If the target game is going to be played on-line then file size is always an issue. You will need to balance the amount of compression with the picture quality to arrive at a compromise that you are happy with. An important consideration of a texture is that the pixel width and height must be a power of 2. Therefore the sides must be of length 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and so on. Texture memory is a rare and wonderful thing and is to be used with care. Textures do not have to be square; they could easily be 16 × 28 or 512 × 2. Deciding on the size for your texture maps involves a balance between the target platform and the largest on-screen size at which the texture is likely to be displayed. Supposing your game is designed to be seen within a browser at 550 × 400 pixels, the texture under consideration is used on the face of your central character and the camera never gets closer to your central character than full figure. This will mean that the face on screen will never exceed 64 pixels square. So there is absolutely no point in having a 512 square texture map, as performance will be noticeably impaired on a slower machine for no visual gain. If your target base-level machine has just 8 Mb of memory on the graphics card then the maximum amount of texture memory you can use is around 1 Mb. At the time of writing our clients usually accept 16 Mb cards as the base level and with these we usually get away with 4 Mb of textures. Texture memory is easily calculated and is not influenced by the compression you use to store your textures. In order to be available to view the textures are uncompressed out of the file. For each texture in your game you need to multiply width times height then you multiply by the render format following Table 18.1.

Suppose you have a 128 × 128 texture that is displayed using #rgba5551 (the default setting); the texture memory used would be 128 × 128 × 2 = 32 768 × 32 K; if you improve the render setting for the texture to #rgba8888 then it would use double the memory. You can see that even small textures use large amounts of memory. The render format gives the amount of bits devoted to each channel. Usually the default setting of #rgba5551 is satisfactory, but this only gives one bit of information to the alpha channel, therefore the texture is either on or off and no anti-aliasing takes place. If you are using a texture that has an irregular-shaped edge that should form the border to the object using this alpha, then changing the setting to either #rgba8888 or #rgba4444 will give a better visual appearance. Always try the lowest setting to see if it gives satisfactory results. Do not default to the highest setting as this will give an unacceptable performance on many machines.

The texture properties are given in Table 18.2.

Table 18.1 How the render format affects texture memory

Render format	Multiplier
#rgba8888	4
#rgba8880	3
#rgba5650	2
#rgba5550	2
#rgba5551	2
#rgba4444	2

Table 18.2 *Texture properties*

Property	Access	Description	Default
name	Get and set	Name of texture	None
type	Get	<i>Possible values</i> #fromfile: bitmap defined as part of 3D import #castmember: bitmap derived from Director cast member	None
member	Get and set	If the type is #castmember, this property identifies the source of the bitmap. If the type is #fromfile, this property is void	None
width	Get	Width, in pixels	None
height	Get	Height, in pixels	None
quality	Get and set	<i>Property with the following possible values</i> #low: texture is not mipmapped #medium mipmapping is at a low bilinear setting (default) #high: the mipmapping is at a high trilinear setting <i>Mip maps are multiple versions of the image at different sizes. If the image was 32×32 then a mipmapped texture would also store 16×16, 8×8, 4×4, 2×2 and 1×1 images. The software would then choose the most suitable size depending on the on-screen size of the texture</i>	#medium
nearFiltering	Get and set	Determines whether bilinear filtering is used when rendering a projected texture map that covers more screen space than the original <i>Bilinear filtering smooths any errors across the texture and thus improves the texture's appearance. Bilinear filtering (a #medium quality setting) smooths errors in two dimensions. Trilinear filtering (a #high quality setting) smooths errors in three dimensions. Filtering improves appearance at the expense of performance, with bilinear being less performance costly than trilinear</i>	TRUE (1)
compressed	Get and set	<i>The property can be TRUE (1) or FALSE (0)</i> TRUE (1): the texture is compressed FALSE (0): the texture is not compressed The value changes automatically from TRUE (1) to FALSE (0) when the texture is to be rendered The value can be set to FALSE (0) to decompress or to TRUE (1) to remove the decompressed representation from memory	TRUE (1)

Shaders

Shaders are one of the most complex areas of real-time 3D. Director allows the developer to manipulate many of the properties at run time. 3DPI is very useful in manipulating a shader to experiment with the effect of each property. Table 18.3 shows the properties of a standard shader. In addition to these properties a shader could be set to be a #painter, #engraver or #newsprint shader; these specialized shaders are useful for creating special effects and a great way to learn about them is to use 3DPI to test applying them. 'Examples/Chapter18/JavaScript04.dir' is a simple

movie that creates a default shader of each type. Open 3DPI and select the ‘Model’ tab. Set the shader using the ‘shaderList’ drop-down menu. In the drop down you should see ‘shd1’, ‘shd2’ and ‘shd3’. These are versions of each shader type created in the ‘beginSprite’ handler using

```
--Lingo
shd1 = w3d.newShader ⌵
    ("shd1", #painter);
shd2 = w3d.newShader ⌵
    ("shd2", #newsprint);
shd3 = w3d.newShader("shd3", ⌵
    #engraver);

//JavaScript
shd1 = w3d.newShader("shd1", ⌵
    symbol("painter"));
shd2 = w3d.newShader("shd2", ⌵
    symbol("newsprint"));
shd3 = w3d.newShader("shd3", ⌵
    symbol("engraver"));
```

Director 3D uses a large number of constants defined in Lingo beginning with the ‘#’ symbol; to use the same constants using JavaScript, ‘#xxx’ becomes ‘symbol(“xxx”)’. More often than not you will be using the ‘standard’ shader. A ‘standard’ shader has a great deal of properties, which you can manipulate. Table 18.3 shows the more commonly used properties.

Model resources

As you know from the previous chapter, when displaying a real-time 3D scene on your screen, the information is stored in many triangles. A model resource is where the triangle information is kept. Lists of triangles that include the vertex, shader and motion information are stored in a manner most suited to a real-time display. For most applications you will use an existing exporter to create your worlds. Exporters are available for most 3D applications and these convert the format for the application into the w3d format. By understanding the format you will understand the problems that are often exhibited when using an exporter. First and foremost is the requirement to convert all the information into a triangular polygonal mesh. A triangle is the most basic polygon and has the huge benefit that it cannot be anything other than planar. Real-time applications use lots of tricks to allow the frames to be painted quickly and an important first consideration is planar triangles. If a polygon is guaranteed to be planar and convex then the order of the vertices on screen defines whether the polygon is facing towards or away from the camera. If you are new to 3D then this may not resonant with the importance that it deserves. But just imagine a scene with 10 000 triangles. This may sound a lot but it is still a very low-polygon scene compared with a TV equivalent.

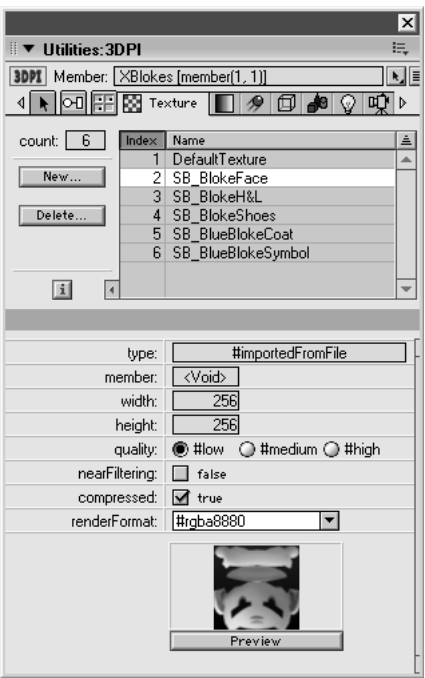


Figure 18.2 Texture properties viewed in 3D Property Inspector

Table 18.3 *Shader properties*

Property	Access	Description	Default
name	Get	The string name of this shader	None
ambient	Get and set	A color object describing the surface's reaction to ambient light	color (63,63,63)
diffuse	Get and set	A color object describing the surface's reaction to diffuse light. Ambient and diffuse color objects together describe a model resource's base color	color (255,255,255)
specular	Get and set	A color object describing the surface's specular highlight color. This setting has an effect only if there are lights in the scene whose specular property is TRUE (1)	color (255,255,255)
shininess	Get and set	An integer between 0 and 100 indicating how shiny a surface is	30.0
emissive	Get and set	A color object describing the color of light this object seems to give off. This does not turn the surface using this shader into a light source; it just gives it the appearance of being one	color (0,0,0)
blend	Get and set	An integer between 0 and 100 indicating how transparent (0) or opaque (100) this surface is. Unlike a texture that includes alpha information, this setting affects the entire surface uniformly	100
transparent	Get and set	This property controls whether or not the model is blended using alpha values or rendered as opaque. The default is TRUE (1) (alpha blended). The functionality of 'shader.blend' is dependent on 'shader.transparent'	TRUE (1)
renderStyle	Get and set	<i>This property can take the following values</i> #fill: the polygon edges of the mesh are filled #wire: the polygon edges of the mesh are rendered #point: the vertices of the mesh are rendered	#fill
flat	Get and set	When shader.flat = TRUE (1), the mesh is given a single color for each polygon. When shader.flat = FALSE (0), the mesh shading is done using Gouraud shading, which tries to smooth out polygon edges	FALSE (0)
textureList[index]	Get and set	A shader can use up to eight layers of textures. Each layer has a specific role. 1: diffuse map 2: light map 3: reflection map 4: gloss map 5: specular map 6–8: reserved	void

Table 18.3 *Continued*

Property	Access	Description	Default
texture	Get and set	This property allows access to the texture for the first layer. It is equivalent to 'textureList[1]' and is supported using Lingo only. An argument of void can be used to disable texturing for the first layer	void
reflectionMap	Get and set	Get: returns the texture associated with the third layer Set: specifies a texture to be used in the third layer and applies the following values textureModeList[3] = #reflection blendFunctionList[3] = #blend blendSourceList[3] = #constant blendConstantList[3] = 50.0	void
diffuseLightMap	Get and set	Get: returns the texture associated with the second layer. Set: specifies a texture to be used in the second layer and applies the following values textureModeList[2] = #diffuse blendFunctionList[2] = #multiply blendFunctionList[1] = #replace	void
specularLightMap	Get and set	Get: returns the texture associated with the fifth layer Set: specifies a texture to be used in the fifth layer and applies the following values textureModeList[5] = #specular blendFunctionList[5] = #add blendFunctionList[1] = #replace	void
glossMap	Get and set	Get: returns the texture associated with the fourth layer Set: specifies a texture to be used in the fourth layer and applies the following values textureModeList[4] = #none blendFunctionList[4] = #multiply	void

To paint the triangles we use a 'z'-buffer. A z-buffer is a very simple concept. It means that we don't have to order the triangles before we start painting them. Remember that the 3D world contains a huge amount of data and we paint this one triangle at a time. If we have to order the triangles to decide which is furthest away before we start painting, then this is likely to take a huge amount of processor power. Instead we just paint the first triangle, but for every point on screen that is painted we also store the distance from camera in another buffer, which is the same size and shape as the visible buffer. Many triangles will overlap with others on screen, so to decide whether to paint an individual pixel we check the z-buffer; if the pixel to be painted is nearer to camera than that already stored in the z-buffer then we paint the pixel, otherwise we ignore it. If we can very

quickly decide that a triangle faces away from camera by the ordering of the vertices then we eliminate anything up to half the polygons in a scene. The calculation of point ordering is very computationally efficient so that is why triangles will be the chosen data format for real-time 3D for a long time to come. Using exporters and Director MX 2004 you are sheltered from the complexities of the data in many ways. However, you are likely to find when developing real-time applications that errors of rendering will occur. If you understand what is happening in the background then you will find it easier to resolve these often frustrating problems.

Another interesting feature of real-time applications is the way that graphics cards operate. A graphics card on a PC is a state machine when it is dealing with triangles. An application sets up how it will deal with triangle data: color, textures, shaders and so on. Then the card is passed vertex data and left to get on with painting the result. For this reason the w3d format separates what you might imagine is a single mesh into multiple meshes based on the shader. Each shader will have a separate mesh set. In some respects this is a less-than-optimum dataset because some point data is stored multiple times. But when rendering a scene it is ideal for the graphics card.

Model resources can be defined in Lingo but for most practical games applications you will create them in a 3D application and then export them. This type of model resource has a type of #fromfile and has the properties shown in Table 18.4.

Although a model resource is a complex dataset, it is not usually practical to manipulate the data. You can achieve this in a limited way by adding a #meshDeform modifier. This allows you to manipulate the vertices by repositioning them. In this way you could create your own bone animation but it is not recommended because performance is far from optimal.

Models

You may have lots of model resources in your world but none of these are visible. Instead they are just a dataset that can be used to visualize a model. Actual models that you see in your world are 'models'; these take a model resource and create an actual instance of the data. It is just like having a class defined that takes an actual instance of the class to use. Models are surprisingly simple having only a few properties that you can manipulate (see Table 18.5).

We will look at using many of these properties in the next chapter.

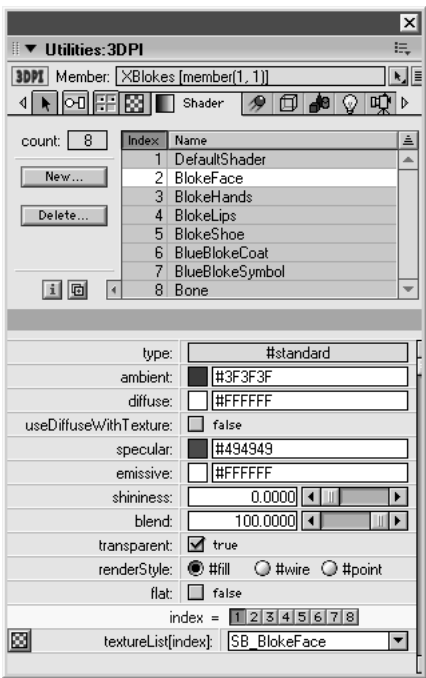


Figure 18.3 Shader properties viewed in 3D Property Inspector

Table 18.4 *Model resource properties*

Property	Access	Description	Range or default
name	Get	Unique string-naming model resource	If imported, the name of the model. If created in Lingo, the assigned name in the constructor function
type	Get	Type of geometry	#plane #box #sphere #mesh #cylinder #particle #fromfile
bone.count	Get	Total number of bones in bones hierarchy	Nonnegative integer
model Resource. getBoneId (“name”)	Get	Returns a unique ID for the bone named ‘name’ in this model’s bone hierarchy. Returns FALSE (0) if no bone by that name can be found	None
vertexList	Get and set	Vector values for each vertex in the mesh. Several faces may share a single vertex	Set the value to the number of vectors specified in your ‘newMesh’ call
normalList	Get and set	Vector values for each normal in the mesh. Several faces may share a single normal. A normalized vector is one in which all components are of unit length. You can use the ‘generateNormals()’ command instead of specifying normals yourself. In that case, set 0 as the number of normals in your ‘newMesh()’ call. The normals are calculated based on a clockwise vertex winding. That is to say, if you imagine the vertices being wound down a spindle, they would be wound from left to right, in a clockwise manner	No default. Instead, set the value to the number of vectors specified in your ‘newMesh’ call
textureCoordinates texcoordlist	Get and set	A list of sublists identifying locations in an image used for texture mapping a triangle. Each sublist contains two values between 0.0 and 1.0, which define a location and can be arbitrarily scaled to any texture size	No default. Instead, set the value to the number of two-element sublists specified in your ‘newMesh’ call

Table 18.4 *Continued*

Property	Access	Description	Range or default
colorList	Get and set	List identifying every color in the mesh. Any color can be shared by several faces. Alternatively, specify texture coordinates for the mesh faces and apply a shader to models using this model resource	No default. Instead, set the value to the number of colors specified in your 'newMesh' call
face.count	Get	Number of triangles in the mesh	The number of faces specified in your 'newMesh' call
face[index].vertices	Get and set	List indicating which vertices to use for faces at designated index points	Set the value to a list of three integers specifying the indexes of the vertices in the 'vertexList' that define this face
face[index].normals	Get and set	List indicating which normals to use for faces at designated index points	Set the value to a list of three integers specifying the indexes of the normals in the 'normalList' that each point of the triangle should use. Don't set a value if you aren't defining your own normals
face [index]. textureCoordinates	Get and set	List indicating which texture coordinates to use for faces at designated index points	Set the value to a list of three integers specifying the indexes of the texture coordinates in the 'textureCoordinates' list that each point of the triangle should use. Don't set a value if you aren't defining your own texture coordinates
face[index].texcoords			
face[index].colors	Get and set	List indicating which colors to use for faces at designated index points	Set the value to a list of three integers specifying the indexes of the colors in the 'colorList' that each point of the triangle should use. Don't set a value if you aren't defining your own colors
face[index].shader	Get and set	Shader used for rendering the face	Shader defined for use with this face

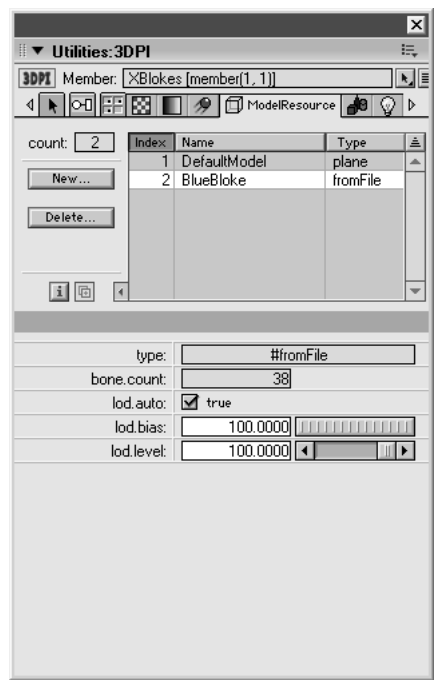


Figure 18.4 Model resource properties viewed in 3D Property Inspector

Motions

Nodes in your 3D world can have motions attached to them. These are often exported from a 3D application. They are stored in the w3d file as a motion resource that can be attached to any node. Usually your chosen exporter will have set the appropriate motion for each object that is moving. A motion has very few properties, simply a name, duration and type. The motion resource used by a bone differs from that used by a model and the type is set accordingly. Bones allow the 3D animator to manipulate a section of a closed mesh allowing for, amongst other things, character animation. A character mesh is amongst the more complex items that need animation in games. You will need to be able to manipulate a subset of the mesh, for example, the arm, while the remainder of the model is either static or animating independently.

Lights

Lights in a 3D world can be as follows:

- *Ambient* the lighting calculation is not concerned with the direction or location of the light, just the strength and color.
- *Directional* the lighting calculation is concerned with the direction and not the location of the light; color is also considered.
- *Point* the lighting calculation is concerned with the location and not the direction of the light; color is also considered.

Table 18.5 *Model properties*

Property	Access	Description	Value
name	Get	Unique string name	Any string
parent	Get and set	This model's parent; either another object or the 3D cast member itself	An object or cast member
child.count	Get	Number of children (but not grandchildren) of a given model	An integer
transform	Get and set	Transform object representing this model's position and orientation relative to its parent's position and orientation: 'transform.position' gives the relative position and 'transform.rotation' gives the relative rotation	Set: a transform object Get: reference to a transform object
userData	Get and set	A property list containing all properties assigned to the model. Users can add, remove, get and set properties on this list	The default list includes the properties assigned in the 3D modeling tool. Additional properties may also be added
resource	Get and set	Model resource object defining model's geometry	Model resource object
shaderList	Get and set	List of all shaders used by the model. Setting this property to a single shader sets every element of the 'shaderList' to that shader	List
shaderList.count	Get	Number of shaders the model uses	Positive integer
shaderList.[index]	Get and set	Provides access to a particular shader used in a specific region of the model	List
shader	Get and set	Provides access to the first shader in the shader list	Shader object
boundingSphere	Get	A list containing a vector and a floating-point value. The vector represents the world position and the value represents the radius of a bounding sphere surrounding the model and all its children	[vector (0,0,0), 0.0]
worldPosition	Get and set	Position of the model in world coordinates. Shortcut for the command node.getWorldTransform().position	Vector object
visibility	Get and set	<i>The way in which the sides of the model's resource are drawn. The choices are</i> #none: in which no polygons are drawn and the model is invisible #front: in which only polygons on the outer surface of the model are drawn, so that, if the camera were inside the model, the model wouldn't be seen. Also known as 'back face culling', this option optimizes performance. #back, in which only polygons on the inside of the object are drawn, so that if the camera were outside the	#none #front #back #both

Table 18.5 Continued

Property	Access	Description	Value
		model, the model wouldn't be seen #both: in which all polygons are drawn and the model is visible regardless of orientation; this may solve drawing problems, but it can also affect performance because twice as many polygons must be drawn The default is #front	
debug	Get and set	Value indicating whether debug information is drawn for this model. If the value is TRUE (1), lines from the 'x'-, 'y'-, and 'z'-axes are drawn sprouting up from the model to indicate its orientation and a bounding sphere is drawn around the model	TRUE (1) or FALSE (0). The default is FALSE (0)
boundingSphere	Get	A list containing a vector and a floating-point value. The vector represents the position of the model in world space and the floating-point value represents the radius of the bounding sphere that contains the model and its children	boundingSphere
worldPosition	Get	Position of the model in world coordinates. A quick shortcut for model.getWorldTransform().position	worldPosition
pointAtOrientation	Get and set	A list of two orthogonal vectors, [objectRelativeDirecton, objectRelativeUp], which control how the model's 'pointAt()' method works	point AtOrientation

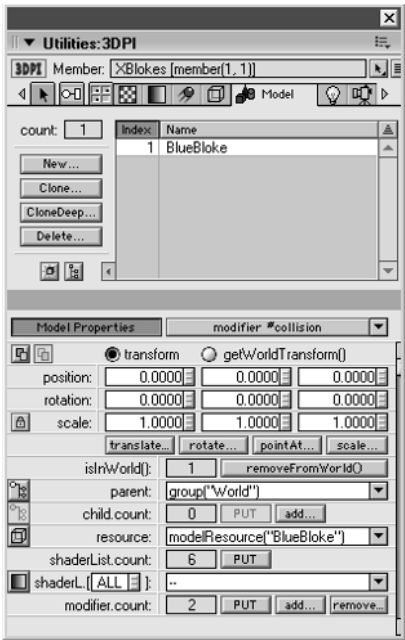


Figure 18.5 Model properties viewed in 3D Property Inspector

Table 18.6 *Motion properties*

Property	Access	Returns
name	Get	Name of motion
duration	Get	Time in milliseconds motion needs to play to completion
type	Get	<i>The type of motion with the following values</i> #keyFrame: suitable for use with keyframe player #bones: suitable for use with bonePlayer #none: no mapping has been made for this motion The default is #none

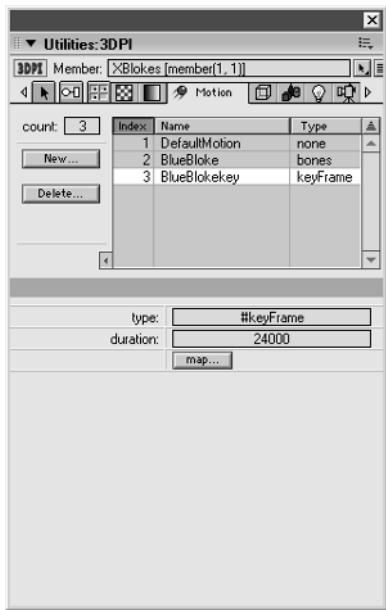


Figure 18.6 *Motion properties viewed in 3D Property Inspector*

- *Spot* the lighting calculation is concerned with both the direction and location of the light; color is also considered.

Usually you will set up your lights in a 3D application, but main properties of a light can be dynamically set using scripting at run time.

Cameras

A camera has the usual properties of a node: 'name', 'parent', 'child.count', 'transform', 'userData', 'boundingSphere' and 'worldPosition'. In addition it has many properties that are very useful when scripting. If you find that your objects disappear as they get near to the camera then you can set 'hither' to a lower value. If they disappear in the distance then set 'yon' to a higher value. If you want to switch between wide angle and telephoto lenses then set the 'projectionAngle'. It is well worth playing with the camera values in a script to get a feel for the effect.

Table 18.7 *Light properties*

Property	Access	Description	Default
name	Get	Unique name of this light. If the light was exported from a 3D modeling package, the name is the name assigned there	None
parent	Get and set	The model, light, camera or group that is this light's parent. If the light has no parent, it cannot contribute light	Group ('world')
child.count	Get	Number of immediate children (no grandchildren) that the light has	0
transform	Get and set	Transform object representing light's position relative to its parent's transform: 'transform.position' gives the relative position and 'transform.rotation' gives the relative rotation	Identity transform
userData	Get and set	A property list associated with this light. The list defaults to the properties assigned in the 3D modeling tool, but users can add or delete properties at any time	Properties assigned in 3D modeling tool
type	Get and set	<i>The kind of light this is. Must be one of the following</i> #ambient: applied to all sides of the model #directional: applied to those parts of the light facing the light's direction. Distance to the light isn't important #point: like a bare light bulb, omnidirectional and illuminating all parts of the model facing the light #spot: like a spotlight, casting light on model parts that face it, with brighter illumination the closer the model is. Similar to #directional, except that the apparent distance from the light is taken into account	None
color	Get and set	Lingo color object defining color and intensity. Ranges from rgb(255,255,255), which is pure white, to rgb(0,0,0), which is no light at all	rgb (191,191,191)
spotAngle	Get and set	Angle of the light's projection cone. If type equals #spot, setting a value less than the umbra causes a Lingo 'property not found' error	90.0
attenuation	Get and set	A three-value vector controlling the constant, linear and quadratic attenuation factors for spotlights	vector (1.0,0.0,0.0)
specular	Get and set	TRUE (1)/FALSE (0) value that controls whether or not the light produces specular effects on surfaces. The property is ignored for ambient lights. Although TRUE (1) is the default, switching to FALSE (0) may improve performance	TRUE (1)

Table 18.7 *Continued*

Property	Access	Description	Default
spotDecay	Get and set	TRUE (1)/FALSE (0) value that controls whether or not spotlight intensity falls off with camera distance	FALSE (0)
pointAtOrientation	Get and set	Two orthogonal vectors (objectRelativeDirection and objectRelativeUp) controlling how the light's 'pointAt' command works	None
boundingSphere	Get	A list containing a vector and a floating-point value, with the vector representing the position and the value the radius of a bounding sphere surrounding the light and all its children	[vector (0,0,0), 0.0]
worldPosition	Get and set	Position of the light in world coordinates. Shortcut for the command 'node.getWorldTransform().position'	Vector object

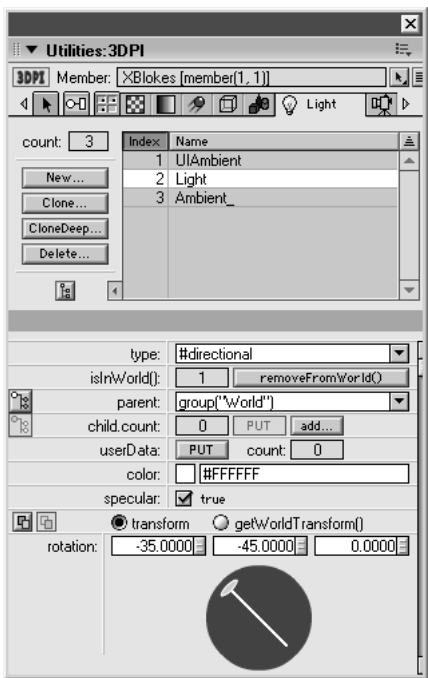


Figure 18.7 *Light properties viewed in 3D Property Inspector*

Check our 'Examples/Chapter18/Lingo05.dir' or 'Examples/Chapter18/JavaScript05.dir' for a sample project that allows you to manipulate some of the camera properties.

When working with cameras, overlays and backdrops are useful elements. A 3D sprite is always best drawn exclusively in its own rectangle. If you allow other sprites to be drawn over the top then

the performance will plummet. For a game you must set the 'directToStage' property of the sprite to TRUE. This is set by default for a 3D sprite. This means that if you want to place a graphic panel in front of the sprite then this must exist in the 3D environment rather than as an alternative sprite channel. You can use Director MX 2004 commands to build a panel then set this panel as the source of a texture which is set as an overlay. For example,

```
--Lingo
tex = member("W3D").newTexture("overlay", #fromCastMember, ⌞
    member("Panel"))
member("W3D").camera[1].addOverlay(tex, point(20, 20), 45)

//JavaScript
var tex = member("W3D").newTexture("overlay", #fromCastMember,
    member("Panel"));
member("W3D").getPropRef("camera", 1).addOverlay(tex, ⌞
    point(20, 20), 45);
```

Check out 'Examples/Chapter18/Lingo06.dir' or 'Examples/Chapter18/JavaScript06.dir' for an example of how to use an overlay. The syntax for a backdrop is very similar and is used in the sample.

Modifiers

Modifiers provide additional control over a mesh. They can be easily added using the 'addModifier()' method. Many exporters will already add some modifiers to a 3D world. The modifiers currently available are shown below.

- *Level of detail* reduces the number of polygons in a model as it is further from the camera.
- *Toon* paints the mesh using flat colors to simulate a cartoon.
- *Inker* adds outlines to a mesh.

Level of detail modifier

The level of detail modifier provides per-model control over the number of polygons used to render a model, by allowing you to reduce the number to a lower value, based on the model's distance from the camera. This modifier is attached to all imported models. The level of detail modifier can work in one of two ways: automatically, using the distance from the camera, or manually. To use manual mode, disable auto mode and then set the properties yourself. Use the properties in Table 18.9 to work with the level of detail modifier.

Toon modifier

The toon modifier changes a model's rendering to imitate a cartoon drawing style. Only a few colors are used and the model's shader, texture and related properties are ignored. Use the properties in Table 18.10 to work with the toon modifier.

Table 18.8 *Camera properties*

Property	Access	Description	Default
name	Get and set	Unique name of this camera. If the camera was exported from a 3D modeling program, the name is the name assigned there	None
parent	Get and set	The model, light, camera or group that is this light's parent. If the camera has no parent, it cannot contribute light	group ('world')
child.count	Get	Number of immediate children (no grandchildren) the camera has	0
transform	Get and set	Lingo transform object representing camera's position relative to its parent's transform. The 'transform.position' property gives the relative position and the 'transform.rotation' gives the relative rotation	Identity transform
userData	Get and set	A property list associated with this camera. The list defaults to the properties assigned in the 3D modeling tool, but users can add or delete properties at any time	Properties assigned in 3D modeling tool
hither	Get and set	A specified distance from the camera that defines the near 'z'-axis clipping of the view frustum. Objects closer than hither are not drawn	5.0
yon	Get and set	A specified distance from the camera that defines the far 'z'-axis clipping of the view frustum. Objects farther than yon are not drawn	3.403e38
rect	Get and set	The rectangle controlling the screen size and position of the camera, with the coordinates given relative to the upper left corner of the sprite	rect(0,0,320,200)
projectionAngle	Get and set	The vertical projection angle of the view frustum	30.0
colorBuffer.clearAtRender	Get and set	TRUE (1) or FALSE (0) value indicating whether color buffer is or isn't cleared out after each frame. If value is set to TRUE (1), the effect is similar to the trails ink effect, although it's limited to the redrawing of models within the sprite itself	TRUE (1)
colorBuffer.clearValue	Get and set	Lingo color object defining color used to clear out buffer if 'colorBuffer.clearAtRender' is TRUE (1)	rgb(0,0,0)

Table 18.8 *Continued*

Property	Access	Description	Default
fog.enabled	Get and set	TRUE (1) or FALSE (0) value indicating whether camera adds fog to the scene	FALSE (0)
fog.near	Get and set	Distance to start of fog	0.0
fog.far	Get and set	Distance to maximum fog intensity	1000.0
fog.color	Get and set	Lingo color object describing fog color	rgb(0,0,0)
fog.decayMode	Get and set	<i>How fog varies between near and far, with the following possible values</i> #linear: density is linearly interpolated between 'fog.near' and 'fog.far' #exponential: 'fog.far' is saturation point; fog.near is ignored. #exponential2: fog.near is saturation point; 'fog.far' is ignored	#exponential
projection	Get and set	Method of determining the vertical field of view, which must be of type #perspective or #orthographic	#perspective
fieldOfView	Get and set	A floating-point value specifying the vertical projection angle in degrees	30.0
orthoheight	Get and set	The number of perpendicular world units that fit vertically into the sprite	200.0
rootNode	Get and set	Property controlling which objects are visible in a particular camera's view. Its default value is the world, so all cameras you create show all nodes within the world. If, however, you change 'rootNode' to be a particular node within the world, a sprite of the cast member will show only the root node and its children	group ('world')
overlay[index].loc	Get and set	Location, in pixels, of the overlay, as measured from the upper left corner of the sprite's rect to the 'regPoint' of 'overlay[index].source'	point(0,0)
overlay[index].source	Get and set	Texture object used as the source for this overlay	None
overlay[index].scale	Get and set	Scale value used by a specific overlay in the camera's list of overlays	1.0
overlay[index].regPoint	Get and set	Texture-relative rotation point, similar to a sprite's 'regPoint'	point(0.0)
overlay[index].rotation	Get and set	Rotation value used by a specific overlay in the camera's list of overlays	0, 0

Table 18.8 *Continued*

Property	Access	Description	Default
overlay[index].blend	Get and set	Blend value used by a specific overlay in the camera's list of overlays. 100 is fully opaque; 0 is fully transparent	100.0
overlay.count	Get and set	Number of overlays in use on this sprite	0
backdrop[index].loc	Get and set	Location, in pixels, of the backdrop, as measured from the upper left corner of the sprite's rect to the 'regPoint' of 'backdrop[index].source'	point(0,0)
backdrop[index].source	Get and set	Lingo texture object used as the source for this backdrop	None
backdrop[index].scale	Get and set	Scale value used by a specific backdrop in the camera's list of backdrops	1.0
backdrop[index].rotation	Get and set	Rotation value used by a specific backdrop in the camera's list of backdrops	0.0
backdrop[index].regPoint	Get and set	Texture-relative rotation point, similar to a sprite's 'regPoint'	point(0,0)
backdrop[index].blend	Get and set	Blend value used by a specific backdrop in the camera's list of backdrops	100.0
backdrop.count	Get	Number of backdrops in use on this sprite	0
boundingSphere	Get	A list containing a vector and a floating-point value, with the vector representing the position and the value representing the radius of a bounding sphere surrounding the camera and all its children	[vector (0,0,0), 0.0]
worldPosition	Get and set	Position of the camera in world coordinates. Shortcut for the command 'node.getWorldTransform()position'	Vector object

Inker modifier

The inker modifier adds silhouette, crease and boundary edges to an existing model. Silhouettes are edges along the border of a model. Crease edges are created when the angle between two areas of the mesh exceeds a given threshold. Use the properties in Table 18.11 to work with the inker modifier.

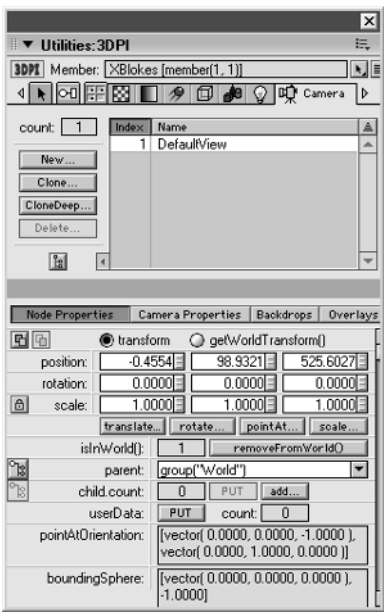


Figure 18.8 Camera properties viewed in 3D Property Inspector

Table 18.9 Level of detail modifier properties

Property	Access	Description	Default
whichModel.lod.auto	Get and set	TRUE (1) means that polygons are automatically reduced based on the distance from the camera. The fewer polygons that are drawn, the faster performance will be. The 'lod.bias' property controls how aggressively this takes place. FALSE (0) means that you can control polygon reduction on a per-model basis, provided you've attached the level of detail modifier to the model. The level of detail modifier lets you override the default settings. To release level of detail data from memory once the model has been streamed in, set the 'userData' property 'sw3D' to TRUE (1)	TRUE (1)
whichModel.lod.bias	Get and set	Aggressiveness with which the level of detail is reduced when in automatic mode. A value of 0.0 is most aggressive and removes all polygons. A value of 100.00 should result in no visible degradation of the geometry. A middle level can be used to remove polygons at run time, which were not removed during authoring	A value between 0.0 and 100.0. The default is 100.0
whichModel.lod.level	Get and set	The percentage of the model resource mesh resolution to use when the automatic mode is FALSE (0)	A value between 0.0 and 100.0. The default is 100.0

Table 18.10 *Toon modifier properties*

Property	Access	Description	Default
whichModel.toon.style	Get and set	<i>The following are the possible values</i> #toon: sharp transitions between available colors #gradient: smooth transitions between available colors #black and white: sharp transitions between black and white	#gradient
whichModel.toon.colorSteps	Get and set	Maximum number of colors available, rounded to the nearest power of 2, with a limit of 16	2
whichModel.toon.shadowPercentage	Get and set	The percentage of color steps to be used in shadows	50
whichModel.toon.highlightPercentage	Get and set	The percentage of color steps to be used in highlight	50
whichModel.toon.shadowStrength	Get and set	A floating-point value that determines shadow darkness	1.0
whichModel.toon.highlightStrength	Get and set	A floating-point value that determines highlight brightness	1.0
whichModel.toon.lineColor	Get and set	Color object describing line color	Black (rgb0,0,0)
whichModel.toon.silhouettes	Get and set	TRUE (1) or FALSE (0) value indicating presence or absence of lines around silhouettes	TRUE (1)
whichModel.toon.creases	Get and set	TRUE (1) or FALSE (0) value indicating whether lines are drawn when mesh boundaries meet at a crease	TRUE (1)
whichModel.toon.creaseAngle	Get and set	A floating-point value controlling crease angle detection	0.01
whichModel.toon.boundary	Get and set	TRUE (1) or FALSE (0) value indicating whether lines are drawn at boundary of surface	TRUE (1)

Table 18.11 *Inker modifier properties*

Property	Access	Description	Default
whichModel.inker.lineColor	Get and set	Color object describing line color	Black (rgb0,0,0)
whichModel.inker.silhouettes	Get and set	TRUE (1) or FALSE (0) value indicating presence or absence of lines around silhouettes	TRUE (1)
whichModel.inker.creases	Get and set	TRUE (1) or FALSE (0) value indicating whether lines are drawn when mesh boundaries meet at a crease	TRUE (1)
model.inker.creaseAngle	Get and set	A floating-point value controlling crease angle detection	0.01
whichModel.inker.boundary	Get and set	TRUE (1) or FALSE (0) value indicating whether lines are drawn at boundary of surface	TRUE (1)

Table 18.12 Subdivision surfaces modifier properties

Property	Access	Description	Default
whichModel.sds.enabled	Get and set	Enables/disables subdivision surfaces modifier functionality	TRUE (1)
whichModel.sds.subdivision	Get and set	<i>The following are the possible values</i> #uniform: mesh is uniformly scaled up in detail, with each face subdivided the same number of times #adaptive: additional detail is added only when there are major orientation changes and only to those areas of the mesh that are currently visible	#uniform
whichModel.sds.depth	Get and set	Maximum recursion depth, with a range of 0 to 5, to which the subdivision surfaces modifier is applied. At a value of 0, no change occurs	1
whichModel.sds.tension	Get and set	Percentage of matching between modified and original surfaces	65
whichModel.sds.error	Get and set	Percentage of error tolerance. This property applies only if 'sds.subdivision' equals #adaptive	0.0

Subdivision surfaces modifier

Subdivision is a technique where a polygon is divided into four polygons. Two levels of subdivision would increase the polygon count 16 fold. The effect is to smooth a mesh at the expense of additional polygons. It is a very useful technique if your character gets close up to the camera. To allow the technique to work an exporter needs to provide additional information at the export stage. Use the properties in Table 18.12 to work with the subdivision surfaces modifier.

Collision modifier

The collision modifier allows a model to be notified of and respond to collisions but the performance is so bad that it's not worth using. Check out the next chapter for other more useful ways to examine your 3D world or use the even more effective Havok Xtra described in Chapter 20.

Animation modifier

When you use most exporters you will find that animation is added to your meshes using modifiers. Director MX 2004 uses two distinct types of animation modifier: keyframe and bonePlayer. Keyframes are suited to a rigid body while bonePlayers deform subsets of a mesh's vertices in different ways and lend themselves to character animations; however with a little ingenuity bonePlayers can be used to animate rigid props in very effective ways.

You are likely to use the current time of an animation modifier in your main game loop to control an animation looping.

Mesh deform modifier

The mesh deform modifier gives you access to the vertex locations of a mesh. You can get and set the vertices in the mesh. It is an essential requirement of using Havok. It is fairly tricky to use but

Table 18.13 *Mesh deform modifier properties*

Property	Access	Description
whichModel.meshDeform. mesh.count	Get	Returns the number of meshes in a model
whichModel.meshDeform. mesh[index].vertexList	Get and set	Returns a list of the vertices for the specified mesh. To modify the vertices in this mesh, set this property to a list of modified vertex positions or modify individual vertices using bracket analysis
whichModel.meshDeform. mesh[index].normalList	Get and set	Returns a list of the normals for the specified mesh
whichModel.meshDeform. mesh[index].textureCoordinateList	Get and set	Returns a list of the texture coordinates for the specified mesh
whichModel.meshDeform. mesh[index].face.count	Get	Returns the number of triangular faces in a given mesh
whichModel.meshDeform. mesh[index].face[index]	Get	Returns a list of three indexes into the vertex: normal, texture coordinate and color lists. These indexes correspond to the corners of the face for the specified mesh
whichModel.meshDeform. mesh[index].face[index]. neighbor[index]	Get	Returns a list of lists describing the neighbors of a particular face of a mesh opposite the face corner specified by the neighbor index (1,2,3). If the list is empty, the face has no neighbors in that direction. If the list contains more than one list, the mesh is nonmanifold. This is rare. Usually the list contains four integer values. The first value is for the index into the mesh[] list, where the neighbor face lives. The second is 'FaceIndex', the index of the neighbor face in that mesh. The third is 'vertexIndex', the index within the neighbor face. The last is 'Flipped', which describes whether the neighbor face is oriented in the same (0) or the opposite (1) way as the original face
whichModel.meshDeform. face.count	Get	Returns the total number of faces in the model, which is equivalent to the sum of all the 'model.meshDeform.mesh[index].face.count' properties in a given model

can give useful water surface effects. Use the properties in Table 18.13 to work with the mesh deform modifier.

Summary

In this chapter we have looked at the many parts of a 3D world. It may all feel rather daunting at the moment and unfortunately 3D can be difficult to get started with for this reason. It is worth persisting however, because Director MX 2004 allows you to create games that would not look out of place on a PlayStation very quickly and as easily as is possible in a 3D environment. In the next chapter we will look at creating some simple working solutions to some of the key problems of a 3D game.

19 3D techniques

After the last two chapters you will have a basic feel for how 3D worlds are created and what data is required to save them to a file. In this chapter we are going to start to look at the scripting side of things. The first step is to get some keyboard inputs to control the camera movement. Relying on direct movement results in a jerky motion so we will look at how to smooth this, by using interpolation techniques that are very easy to use. Then we will use the knowledge gained to follow an object. The triangles in your 3D world that make up your objects know nothing about collisions and will happily intersect each other. A very important technique in 3D game development is to avoid this without creating a significant performance hit; in this chapter we will examine some key strategies for this that you can implement in your own games. Finally we will look at how you can use the 'pointAt' methods to align an object to an uneven terrain. Lots to learn but great fun to play with. This is a chapter that is very definitely best read while at your computer playing with the sample programs.

Moving the camera

Nearly all 3D games will require a moving camera. The simplest moving camera example is given in 'Examples/Chapter19/Lingo01.dir'. In this example we add a behavior script to the 3D sprite. The behavior is shown in Listing 19.1. Lines 3–7 are a simple initialization method. First we store the 3D member in a property variable for convenience. Next we reset the 3D world; this a recommended method for all 3D members in your cast, as strange errors will often result from neglecting to reset every 3D world. You have been warned! In line 6 we assign the main camera to a property variable. Then the main action goes into an 'enterFrame' handler, which is called as many times a second as the tempo dictates. The handler checks each of the arrow keys using the '_key' objects 'keypressed()' method. This is passed a code which corresponds to a key on the keyboard; if the key is pressed then the method returns 'true' and if not then 'false' is returned. The left and right arrows are used for rotation and the up and down arrows for movement. The simplest rotation method is defined in line 12. Here three numeric values are passed. These values correspond to the rotation around the 'x'-, 'y'- and 'z'-axes. An x rotation will tilt forward and back (pitch), y rotation turns left and right (heading) and z rotation banks. 3D rotations expressed in this way are difficult to predict when more than one axis is used. The reason for this is that the order of the rotations affects the final result. Tilting forward, then rotating left and finally banking will give a different orientation to rotating first, then banking and then tilting. If you read the literature, this method is known as Euler angles and is best left to single-axis rotations. Translation is along the x-, y- and z-axes. Line 20 moves the camera 10 units forward along its z-axis.

```
1 property pW3d, pCam
2
```

```

3  on beginSprite
4    pW3d = member("Boxes")
5    pW3d.resetWorld()
6    pCam = pW3d.camera[1]
7  end
8
9  on enterFrame
10   if (_key.keypressed(123)) then
11     --Left Arrow
12     pCam.rotate(0,2,0)
13   end if
14   if (_key.keypressed(124)) then
15     --Right Arrow
16     pCam.rotate(0,-2,0)
17   end if
18   if (_key.keypressed(125)) then
19     --Down Arrow
20     pCam.translate(0,0,10)
21   end if
22   if (_key.keypressed(126)) then
23     --Up Arrow
24     pCam.translate(0,0,-10)
25   end if
26 end

```

Listing 19.1

Figure 19.1 shows the effect of running the program and pressing the up arrow followed by the left arrow. Because the camera is tilted down, a movement along its 'z'-axis moves the camera forward and down. In many cases you will want to fix the camera height. Another problem with this simple technique is that the rotation builds on an already tilted camera so instead of rotating and maintaining a camera that appears to be upright the camera banks as well as looking left.

Listing 19.2 shows a quick way to improve the camera motion. In this example we use an alternative rotate method. Line 12 shows how to use angle/axis rotation. We pass a world position. In this instance the camera's position, the second parameter, is the angle to use for the rotation; here we use a vector that points directly up. The third parameter defines how much rotation to apply and finally we pass a parameter defining which coordinate space to use for the rotation. Using this technique we control which way is up and as you can see from Figure 19.2 uprights remain vertical. To ensure that we maintain a constant height we use lines 20–22. Here the camera height ('y') is stored before the translation, then we do the translation and finally we put the camera back to the original height. Using this technique we can use the translate method while remaining in control of the camera's height.

```

1  property pW3d, pCam
2

```

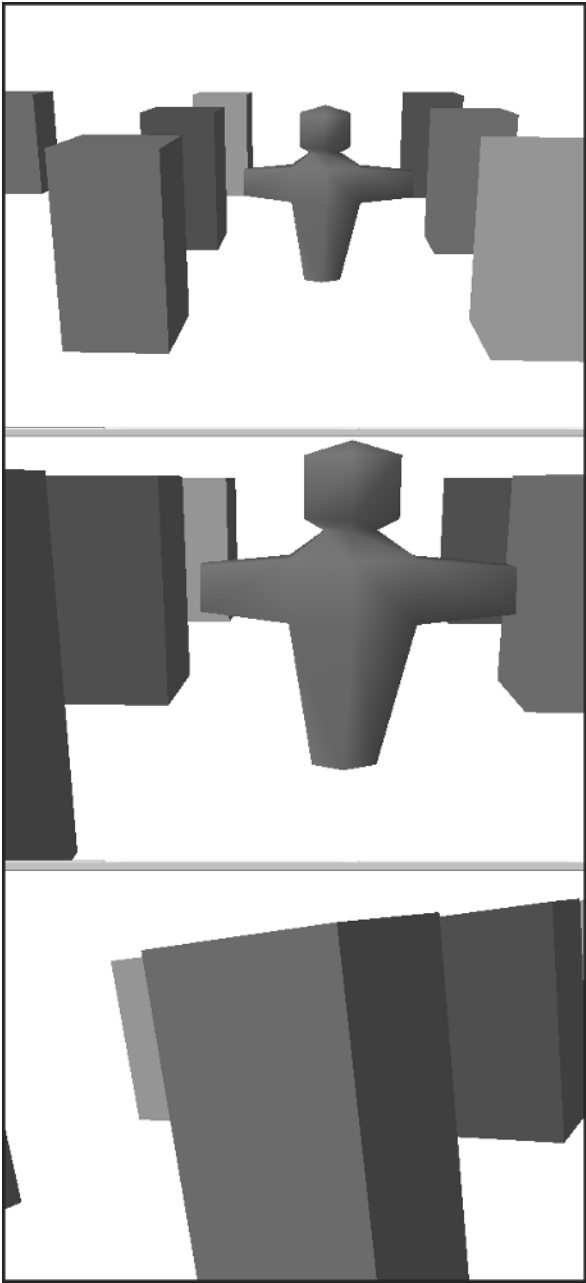


Figure 19.1 *Examples/Chapter19/Lingo01.dir*

```

3  on beginSprite
4    pW3d = member("Boxes")
5    pW3d.resetWorld()
6    pCam = pW3d.camera[1]
7  end
8
9  on enterFrame
10   if (_key.keypressed(123)) then
11     --Left Arrow
12     pCam.rotate(pCam.worldPosition, vector(0,1,0), 2, #world)
13   end if
14   if (_key.keypressed(124)) then
15     --Right Arrow
16     pCam.rotate(pCam.worldPosition, vector(0,1,0), -2, #world)
17   end if
18   if (_key.keypressed(125)) then
19     --Down Arrow
20     y = pCam.worldPosition.y
21     pCam.translate(0,0,10)
22     pCam.worldPosition.y = y
23   end if
24   if (_key.keypressed(126)) then
25     --Up Arrow
26     y = pCam.worldPosition.y
27     pCam.translate(0,0,-10)
28     pCam.worldPosition.y = y
29   end if
30 end

```

Listing 19.2

Figure 19.2 shows that the second listing has a much improved camera motion. The problem is that the motion has no inertia. Commercial games will have a camera that accelerates and decelerates. This technique is easy to add to your games and well worth the small additional effort.

Improving the camera motion

The trick with creating smoother motion of the camera is to avoid moving the camera directly at all. Instead, create a null object, an object that has no geometry but has a place in the world. The position and orientation of an object in your 3D world are defined using a transformation matrix. If you are interested in the details of the transform matrix then take a look at Appendix A where some of the gory details of the maths involved are outlined. If you are happy to leave it as a black box that does its magic then that's fine for most applications. What we do in 'Examples/Chapter19/Lingo03.dir' is create a null object at line 7 then place it at exactly the same position and orientation

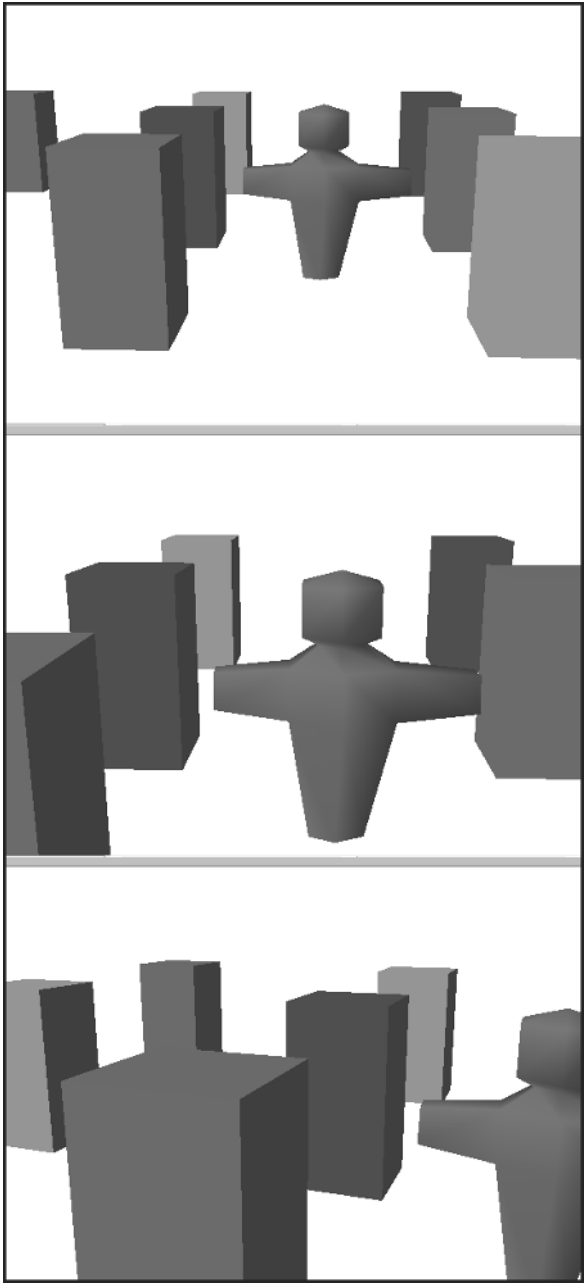


Figure 19.2 *'Examples/Chapter19/Lingo02.dir'*

as the camera by copying the camera's transform into the transform of the new object, line 8. Now instead of moving the camera we move this null object. So how does the camera move? Surely we will just have a static camera. If you thought that then go to the top of the class because you obviously understand the concepts very well. The second trick is the key to this technique and is shown in line 32. Director MX 2004 has a wonderful method of a transformation matrix, which allows you to blend one with another using the 'interpolateTo' method. This method takes two parameters; first the transformation matrix that you are blending to and second the percentage of this matrix to use in the final result. In this example we use 90% of the original and 10% of the null object. So it takes several redraws after the null object has stopped before the camera catches up. Try altering the percentage parameter and you will see that the inertia of the camera is affected. Using this method greatly improves the camera motion and is easily implemented.

```

1  property pW3d, pCam, pCamNull
2
3  on beginSprite
4      pW3d = member("Boxes")
5      pW3d.resetWorld()
6      pCam = pW3d.camera[1]
7      pCamNull = pW3d.newModel("CameraNull")
8      pCamNull.transform = pCam.transform.duplicate()
9  end
10
11 on enterFrame
12     if (_key.keypressed(123)) then
13         --Left Arrow
14         pCamNull.rotate(pCamNull.worldPosition, vector(0,1,0), ↵
15             2, #world)
16     end if
17     if (_key.keypressed(124)) then
18         --Right Arrow
19         pCamNull.rotate(pCamNull.worldPosition, vector(0,1,0), ↵
20             -2, #world)
21     end if
22     if (_key.keypressed(125)) then
23         --Down Arrow
24         y = pCamNull.worldPosition.y
25         pCamNull.translate(0,0,10)
26         pCamNull.worldPosition.y = y
27     end if
28     if (_key.keypressed(126)) then
29         --Up Arrow
30         y = pCamNull.worldPosition.y
31         pCamNull.translate(0,0,-10)

```



```

30     pCamNull.worldPosition.y = y
31   end if
32   pCam.transform.interpolateTo(pCamNull.transform, 10)
33 end

```

Listing 19.3

Following an object with the camera

Tracking an object is quite easy. 'Examples/Chapter19/Lingo04.dir' shows one method. In the 'beginSprite' handler we initialize another useful property variable 'pFigure' (line 9). Then we add the camera null object to this as a child. A child will move with its parent by default. But we need to make sure that as it is attached to the parent it doesn't move. Line 10 shows the use of 'preserveWorld' to ensure that the position of the camera null is unaltered as it becomes a child of the model 'Figure'. Now we have a model that is positioned at the camera but that is fixed to the model Figure almost as though a rod attaches the two together. The 'enterFrame' handler uses the same key presses but now we are moving the Figure model. Moving the Figure will cause the camera null to move. One thing to watch out for when a model has a parent is the use of its transform. An object's transform represents its position and orientation in relation to its parent. In this instance we need its position and orientation in world terms. Fortunately there is a simple technique for this, which uses the method 'getWorldTransform()'. Line 30 uses this method and stores the result in the local variable 't'. It is this transform that is passed to the 'interpolateTo' method. Try replacing this line with the direct use of 'pCamNull.transform' to see that nothing will happen. The reason for this is the camera null is not moving in relation to its parent so the transform is unchanged by any motion of the Figure. Remember that a parented object acts as though a rigid rod connects the object to its parent. You may find this a little confusing but it is an important point so here is an actual example to help clarify the issues.

Supposing that the camera null is at world position (100, 200, 300) and the 'Figure' is at (50, 100, 75). When the camera null is parented to the Figure, the position of the camera null will be in relation to the Figure so its position will be changed to (100-50, 200-100, 300-75) or (50, 100, 225). The camera null does not move because the software knows that a parented object is described in relation to its parent so when it positions the camera null it first moves it to the position of its parent then moves it again the amount defined in the transformation matrix of the object. As the Figure moves around, the camera null moves with the Figure but the value described in its own transformation matrix remains fixed at (50, 100, 225). If we use this value directly for the interpolation then no movement will result. But by getting the transformation matrix of the camera null described in world coordinate space using the 'getWorldTransform()' method we can quickly convert between parent space and world space. If you are still confused then I suggest reading the above again until the difference between parent space and world space is fully understood.

```

1  property pW3d, pCam, pCamNull, pFigure
2
3  on beginSprite
4    pW3d = member("Boxes")

```

```

5    pW3d.resetWorld()
6    pCam = pW3d.camera[1]
7    pCamNull = pW3d.newModel("CameraNull")
8    pCamNull.transform = pCam.transform.duplicate()
9    pFigure = pW3d.model("Figure")
10   pFigure.addChild(pCamNull, #preserveWorld)
11 end
12
13 on enterFrame
14   if (_key.keypressed(123)) then
15     --Left Arrow
16     pFigure.rotate(pFigure.worldPosition, vector(0,1,0), ↺
17       2, #world)
18   end if
19   if (_key.keypressed(124)) then
20     --Right Arrow
21     pFigure.rotate(pFigure.worldPosition, vector(0,1,0), ↻
22       -2, #world)
23   end if
24   if (_key.keypressed(125)) then
25     --Down Arrow
26     pFigure.translate(0,0,10)
27   end if
28   if (_key.keypressed(126)) then
29     --Up Arrow
30     pFigure.translate(0,0,-10)
31   end if
32   t = pCamNull.getWorldTransform()
33   pCam.transform.interpolateTo(t, 10)
34 end

```

Listing 19.4

Basic collision testing

Although Director MX 2004 provides a collision modifier, it is not recommended for games applications, because the performance of your games will plummet. Collision detection in games is always going to be a compromise. The simplest method is to test for bounding sphere distances. But this is so inaccurate that it is not usually a practical approach and if the number of objects in your scene is high then it can be quite processor intensive. The route we use on all 3D games is to cast a ray out from the model and see if the ray hits anything; if it does then we query the distance to the hit and if it is below a certain bound then we either stop the object from moving, move it back or deflect it. Listing 19.5 from 'Examples/Chapter19/Lingo05.dir' shows the first option.

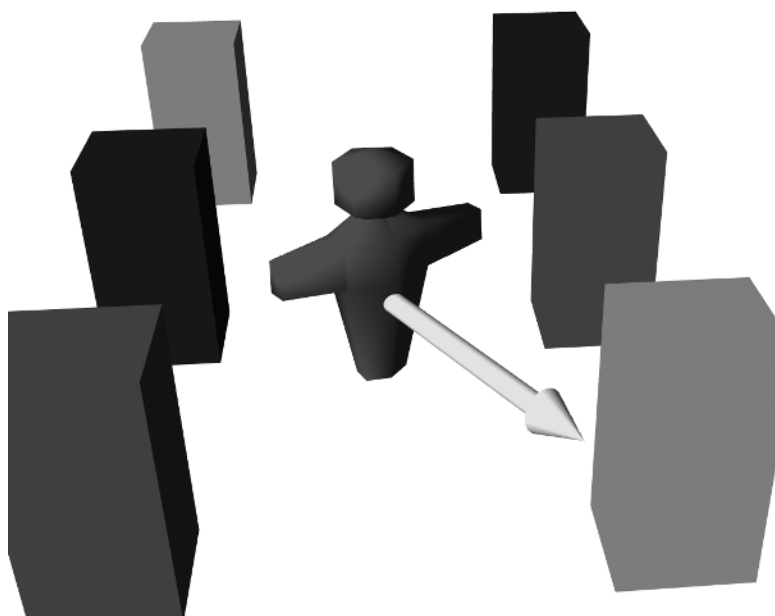


Figure 19.3 *Illustrating the 'modelsUnderRay' method*

The method to use is 'modelsUnderRay'. Don't be deceived by the use of the word 'Under' into thinking that this only ever applies to rays that point down. A ray can point in any direction. Figure 19.3 shows the direction of the ray used in the 'checkForwards' handler shown in Listing 19.5, lines 36–50.

Let's examine how the 'modelsUnderRay' method works. The method applies to a world and in the case shown at line 37 of Listing 19.5 the world is stored in the property variable 'pW3d'. The method can be used in two ways; we will be looking at the more complex but massively more useful method. The simpler method just detects whether anything gets hit by the ray and returns a list of models that are hit by the ray. Because no distance information is provided the simple version is rarely useful enough to be relied on as a collision detection option although it is suitable for a gun shot where simply being in the direction of the ray is sufficient for the model to be destroyed.

The more complex form takes four parameters: first the place in your world to start the ray cast, second the direction (in this example we cast the ray down the 'z'-axis of the 'Figure'), third the number of intersections we are interested in (in this case 2) and finally a constant that tells the method we are interested in getting detailed information about the intersection. The method returns a list; if this is empty then no models are hit by the ray; if the list is not empty then line 39 will result in a true value so we then get to line 40 where we exclude intersections with ourselves using the syntax shown in line 41. If the model is not the Figure then the intersection is important. One of the properties in the list returned is the distance from the start of the ray cast to the intersection; we test whether this is less than 100 in line 43. If so then we return false, if not then true is returned from the handler. The handler is used at line 30 before the Figure is translated. Translation only occurs if the handler returns true, therefore the ray cast does not hit a model nearer than 100 units. For many games this is sufficient to allow your characters to run around a 3D world and not

walk through walls! But in this instance a glancing blow will still result in the ‘arms’ of the Figure going through the blocks. Each game will need you to develop custom solutions. In this example you could cast from the end of both arms and only allow forward or back movement when both are clear. The ‘checkForwards’ handler is mirrored by the ‘checkBack’ handler where a ray is cast back from the Figure. This is used when moving in reverse.

```

1  property pW3d, pCam, pCamNull, pFigure, pInfo
2
3  on beginSprite
4    pW3d = member("Boxes")
5    pW3d.resetWorld()
6    pCam = pW3d.camera[1]
7    pCamNull = pW3d.newModel("CameraNull")
8    pCamNull.transform = pCam.transform.duplicate()
9    pFigure = pW3d.model("Figure")
10   pFigure.addChild(pCamNull, #preserveWorld)
11   pInfo = member("Info")
12   pInfo.Text = "No collision detected"
13 end
14
15 on enterFrame
16   if (_key.keypressed(123)) then
17     --Left Arrow
18     pFigure.rotate(pFigure.worldPosition, vector(0,1,0), ↺
19       2, #world)
19   end if
20   if (_key.keypressed(124)) then
21     --Right Arrow
22     pFigure.rotate(pFigure.worldPosition, vector(0,1,0), ↻
23       -2, #world)
23   end if
24   if (_key.keypressed(125)) then
25     --Down Arrow
26     if (checkBack()) then pFigure.translate(0,0,10)
27   end if
28   if (_key.keypressed(126)) then
29     --Up Arrow
30     if (checkForward()) then pFigure.translate(0,0,-10)
31   end if
32   t = pCamNull.getWorldTransform()
33   pCam.transform.interpolateTo(t, 10)
34 end
35

```

```

36 on checkForward
37   col = pW3d.modelsUnderRay(pFigure.worldPosition,
38                               pFigure.transform.zAxis, 2, ↵
                               #detailed)
39   if col.count>0 then
40     repeat with i=1 to col.count
41       if (col[i].model <> pFigure) then
42         pInfo.Text = "Distance to collision is " & col[i]. ↵
                     distance
43         if col[i].distance < 100 then return false
44       end if
45     end repeat
46   else
47     pInfo.Text = "No collision detected"
48   end if
49   return true
50 end
51
52 on checkBack
53   col = pW3d.modelsUnderRay(pFigure.worldPosition,
54                               pFigure.transform.zAxis, 2, ↵
                               #detailed)
55   if col.count>0 then
56     repeat with i=1 to col.count
57       if (col[i].model <> pFigure) then
58         pInfo.Text = "Distance to collision is " & col[i]. ↵
                     distance
59         if col[i].distance < 100 then return false
60       end if
61     end repeat
62   else
63     pInfo.Text = "No collision detected"
64   end if
65   return true
66 end

```

Listing 19.5

Aligning to a terrain

A common problem in game development is to orientate a model to an undulating terrain. A simple method of doing this is to use the 'pointAt' method. pointAt takes two vectors (forward and up) and attempts to align the model so the forward and up orientation of the model is aligned to these two vectors. You set what is regarded as forward and up using the 'pointAtOrientation' property.

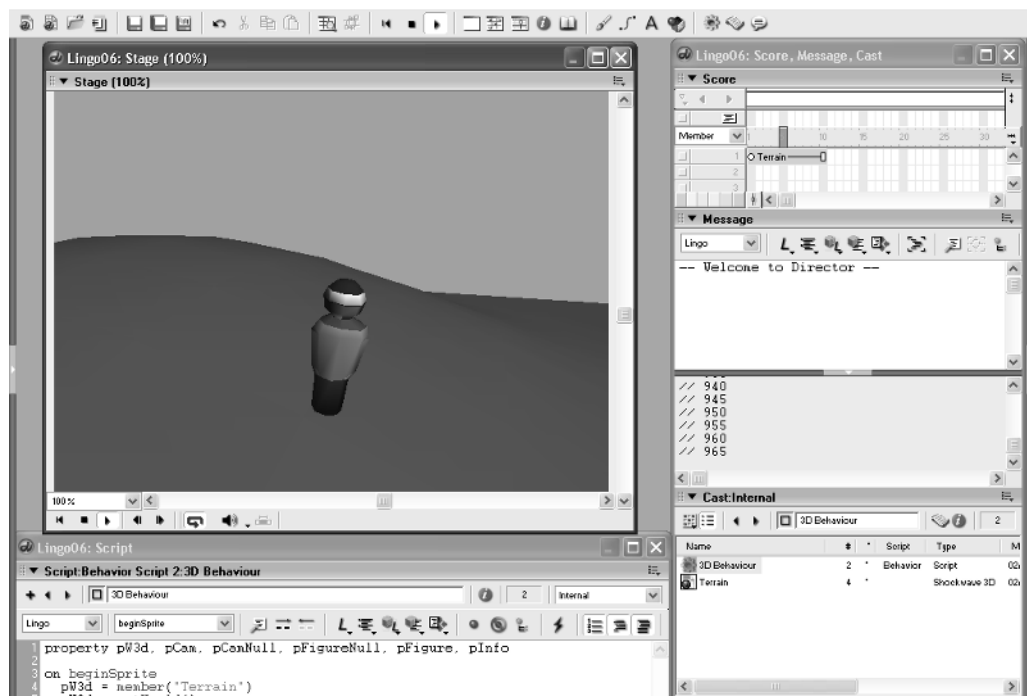


Figure 19.4 'Examples/Chapter19/Lingo06.dir' in development

Line 10 of Listing 19.6 shows how the 'pointAtOrientation' of the 'Figure' model is set. Forward is set as directly up the 'y'-axis and up is defined as straight down the 'z'-axis. Usually we have been using forward as down the z- and up as along the y-axis. In this we flip the values, because we want the object to point at somewhere up a normal that sticks out of the terrain mesh and to keep the direction set by aligning the z-axis. If we had the forward vector being down the z-axis and the up vector along the y-axis then we could point at somewhere ahead of us. But terrain alignment is done by pointing at something above us and this should affect the y-axis of the model. You can keep changing the pointAtOrientation vectors to allow you to get the maximum value from the 'pointAt' method.

```

1  property pW3d, pCam, pCamNull, pFigureNull, pFigure, pInfo
2
3  on beginSprite
4    pW3d = member("Terrain")
5    pW3d.resetWorld()
6    pCam = pW3d.camera[1]
7    pCamNull = pW3d.newModel("CameraNull")
8    pCamNull.transform = pCam.transform.duplicate()
9    pFigure = pW3d.model("Figure")
10   pFigure.pointAtOrientation = [vector(0,1,0),vector(0,0,1)]

```

```

11 pFigureNull = pW3d.newModel("FigureNull")
12 pFigureNull.transform = pFigure.transform.duplicate()
13 pFigureNull.addChild(pCamNull, #preserveWorld)
14 end
15
16 on enterFrame
17   if (_key.keypressed(123)) then
18     --Left Arrow
19     pFigureNull.rotate(pFigureNull.worldPosition, vector(0,1,0),
20                       2, #world)
21   end if
22   if (_key.keypressed(124)) then
23     --Right Arrow
24     pFigureNull.rotate(pFigureNull.worldPosition, vector(0,1,0),
25                       -2, #world)
26   end if
27   if (_key.keypressed(125)) then
28     --Down Arrow
29     pFigureNull.translate(0,0,1)
30   end if
31   if (_key.keypressed(126)) then
32     --Up Arrow
33     pFigureNull.translate(0,0,-1)
34   end if
35   alignToTerrain()
36
37   t = pCamNull.getWorldTransform()
38   pCam.transform.interpolateTo(t, 10)
39 end
40
41 on alignToTerrain
42   tpos = pFigureNull.worldPosition
43   tpos.y = 100
44   tdata = pW3d.modelsUnderRay(tpos, vector(0, -1, 0), 2, #detailed)
45
46   if tdata.count then
47     tcount = tdata.count
48     repeat with i = 1 to tcount
49       if tdata[i].model <> pFigure then
50         pFigureNull.worldPosition = tdata[i].isectPosition
51         pFigure.transform.interpolateTo(pFigureNull.getWorldTransform(),
52                                         10)
53       --Orientate to set

```

```

54         tfront = pFigure.transform.zAxis
55         tpos = pFigure.worldPosition + (10.0 * tdata[i].isectNormal)
56         startT = pFigure.getWorldTransform()
57         pFigure.pointAt(tpos, tfront)
58         endT = pFigure.getWorldTransform()
59         pFigure.transform = startT.interpolate(endT, 10)
60         exit repeat
61     end if
62 end repeat
63 end if
64 end

```

Listing 19.6

The main work in this example is done by the ‘alignToTerrain’ handler. In line 42 we get the world position of the ‘Figure’. In this example we use a null object that is positioned on the terrain but not rotated to align with the terrain because this would affect the camera motion, which relies on an object where upright is always directly up the ‘y’-axis. The Figure object that we see uses information from the Figure null object and information from the terrain alignment. In line 43 we set the ray cast start position to be well above the terrain by setting the y value to 100; our highest hill is less than half that value. Line 44 uses the important ‘modelsUnderRay’ method to derive data for the terrain mesh. We cast directly down and return two models in a detailed format. If any models are returned then line 47 will evaluate to true and so the repeat loop (lines 48–62) will be executed. Line 50 positions the null at the intersect position of the ray and the terrain (line 51) interpolates the current Figure position in the direction of the null. At line 54 we store the current ‘z’-axis of the Figure and create a point that is directly along the intersect normal (think of a normal as a stick that points directly out from the mesh, so its direction will vary all around this undulating mesh) from the intersection point. Then we store the current transformation matrix of the Figure model in line 56. In line 57 we use the ‘pointAt’ method to align the mesh to the terrain. Then we blend the two orientations using the now familiar ‘interpolateTo’ method. It is vital to use an interpolation technique because with low-polygon meshes like this the different orientation of each triangle will result in a very jerky motion. Try commenting out lines 58 and 59 to see how the interpolation technique has smoothed out the bumps.

Summary

3D can often seem daunting. Most things are best learnt in small, easily understood sections. In this chapter we have looked at lots of simple but effective methods to use in your own games. Camera control is an important topic as is collision detection. Hopefully you now have sufficient grounding to develop on the topics presented and add your own magic. We have covered a lot of ideas and if the techniques have been fully understood then you are ready for the final chapter of the book where we create a fully working snowboarding game using a combination of the techniques illustrated and the additional functionality of the fantastic Havok extra.

20 Using Havok

For most games that you create using Shockwave 3D you will benefit by controlling at least some of the interactions using Havok. This remarkable Xtra is a rigid body physics simulation. If that sounds like a complicated thing and your eyes are beginning to cloud over then wake up because this is one to try. Trust me on this; you need to know this stuff. It's tricky at first but the results are nothing short of amazing and before long you will be creating physics-based games with the best of them. In this chapter we will start with an introduction to physics simulations followed by an overview of Havok in particular. We will look at a simple example, then a more complex example that adds more control into the scene. Finally we will look at applying this to an actual snowboarding game that is available on-line. It's another chapter with lots to learn but once you have mastered the basics you will find that Havok looks after things in your game that even after weeks of complex coding would otherwise never have looked right.

What is rigid body physics?

The ideas used in a rigid body simulation are the work of two great mathematicians who lived nearly 400 years ago. Most of the work is based on Newtonian mechanics. You may have come across it in school. Sir Isaac Newton was born in the UK in 1643 and in studies he made while at university he came up with simple laws that express the motion of objects. The other major figure is Robert Hooke, born in 1635; he gave the world a very simple mathematical model of spring behavior, which is also used in the Havok Xtra.

A rigid body physics simulation takes the meshes in your scene, gives them weight and friction then if one hits another it will react in way that is a close mirror to what happens in real life, except no deformations of the object will occur. The bodies will always hold their shape, but they may go tumbling off into the distance. For most 3D worlds the actual size of the world is not important. However when you start playing with physics, size is very important. Units in Havok are by default expected to be in meters and kilograms. The usual value for the force of gravity is 10 ms^{-2} , which means that the object will be traveling at 10 ms^{-1} after 1 s, 20 ms^{-1} after 2 s and $(n \times 10) \text{ ms}^{-1}$ after n s. If you have an object that is about 1 m^2 and give it a weight of around 10 kg and then drop it from a height of 3 m with a force of gravity of 10 m^{-2} , the way it tumbles will be just how you anticipated; it will hit the floor in less than half a second and bounce or tumble depending on settings that you apply to the object to initialize it. If however, the object is 100 units square and is dropped from 300 units high with a similar gravitational force, but due to the camera position the object is believed to be the size of a crate, then you will think that the speed is wrong; in actual fact you are dropping a large warehouse from 300 m in the sky, not a small crate. The object will take

$$10 + 20 + 30 + 40 + 50 + 60 + 70 = 280$$

Distance fallen each second

Total for 7 s.

As you can see from the simple calculation, even after 7 s the ‘crate’ will not have hit the floor. This is not because the physics simulation is ‘wrong’, the problem is scale. In actual fact the distance traveled (d) is given by the formula

$$d = vt + 1/2at^2,$$

where t = time. Because v (starting velocity) is zero, a (acceleration) is 10 and d is 300 we can juggle the equation to get

$$t = \sqrt{60} = 7.746 \text{ s},$$

whereas when the distance to travel is just 3 m the formula gives the answer 0.775 s.

It is possible to match the scales of your geometry with the way the world behaves using Havok; the important first step is to get the scales balanced.

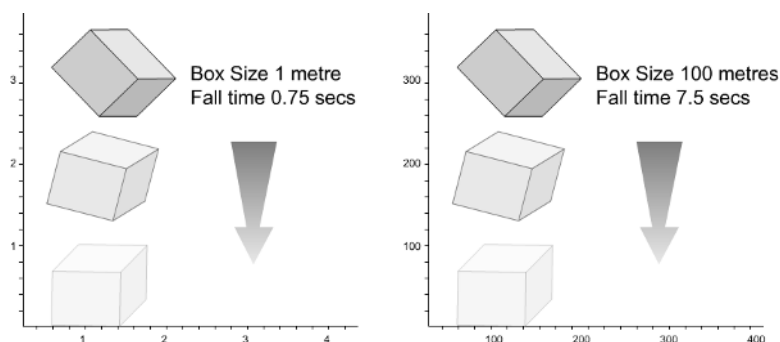


Figure 20.1 *Scale in a physics simulation*

Because a rigid body physics simulation has to work at an alarming rate it approximates a huge number of the calculations to speed up the processing. Nevertheless you should never give a physics simulation more work to do than it needs. Suppose you are using a physics simulation to move a car that is made from 1200 polygons. The car will look beautiful but the same motion can be achieved by using a very simple substitute for the car. Look at Figure 20.2 where we show a car and the proxy object that can be used by the physics simulation to move the car. The proxy contains less than 100 polygons, whereas the car contains several thousand. The technique is to make the proxy invisible, to let Havok move the proxy and then to copy the transform matrix of the proxy to the actual visible car model. A rigid body simulation at some stage in the calculations gets down to examining each vertex and triangle in the object. If we can get these down then the performance will soar.

A common problem with a physics simulation is the time interval used allowing impossible things to happen. Take a look at Figure 20.3 where we see a ball going straight through a wall when it should have bounced off. Havok knows nothing about your world in the way you imagine it. It does know about time. If the number of frames updated per second is 25 and Havok calculates the positioning of the ball just prior to hitting the wall at 1.0 seconds, then just 0.04 seconds later

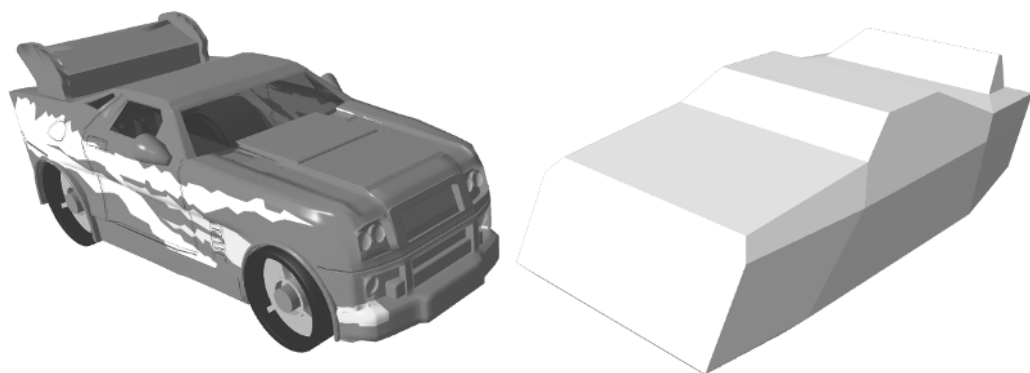


Figure 20.2 *Using a proxy object to improve the performance*

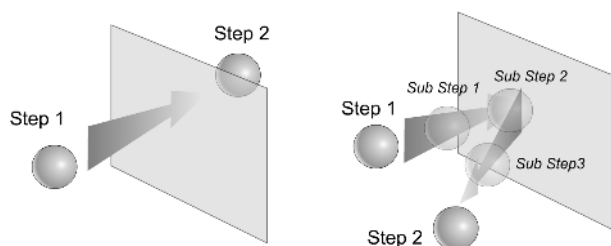


Figure 20.3 *A ball passes magically through a wall*

that the ball is happily on the opposite side of the wall from where it appeared just a short interval before, as far as Havok is concerned that is OK. It does not ‘realize’ that the ball passed through the wall, because in the intervals of time where it does do calculations the ball is found to be in an acceptable position. There is a way out of this problem that does not require a computer that is capable of displaying 200 frames a second. Instead we tell Havok to calculate substeps. If we tell it to calculate three substeps then the very different behavior illustrated on the right of Figure 20.3 will occur. Although only two screen updates occur, three additional intermediary calculations ensure that Havok is kept abreast of the situation. It is possible to adjust the number of substeps to suit your application. You should aim to use the minimum number of substeps that avoids physics errors. Each additional substep takes computational time and therefore by minimizing the number you will get the optimum frame rate performance.

An overview of the Havok Xtra

In this section we will look at some key properties and methods of the Havok Xtra.

Havok object properties

- *havok.initialized* – this property allows you to test whether Havok is currently ready to use. It can be tested but cannot be set directly.

- *havok.tolerance* – you will learn the significance of tolerances as you initialize your first Havok scene. For now just be aware that this can be accessed but not set directly.
- *havok.scale* – we have discussed how important it is to balance the scales of your world to the standard values of 1 meter and 1 kilogram. Again this is down to using the initializing method so this property is access only.
- *havok.timeStep* – this property holds the current time-step factor for the simulation. The time-step factor represents the amount of time that the physics simulation advances with each call to ‘havok.step()’. It can be both accessed and set.
- *havok.subSteps* – this property holds the current number of substeps used by Havok during each call to ‘havok.step()’. It can be both accessed and set.
- *havok.gravity* – this property holds the current force of gravity for the simulation as a vector. You specify gravity display units, so you need to be careful when setting it up. If using a scale factor of 1.0 (that is, meters) then gravity should be (0, -9.81, 0) to act appropriately (assuming positive ‘y’ is up).
- *havok.rigidBody* – this is a list of all rigid bodies in the simulation.

Havok object methods

havok.initialize(W3DMember, tolerance, worldScale)

In many ways the initialize method is the most important, as it starts the simulation and the parameters passed greatly affect the appearance of the simulation. There are two ways to define physical information for the Havok Xtra. The first method for creating physical simulation information is through a modeling tool. The modeling tool that you use must support exporting .hke files. You can import this .hke file as a movie cast member using the ‘File>Import’ menu options. .hke files already contain world scaling information and tolerance (as specified within the 3D modeller), so you do not have to supply this value when initializing. 3DS supports .hke files but Lightwave does not. So in the examples for this chapter we will use the second method for creating physical information where we use the models within a 3D scene to define the physics information. In this case, you must create a blank Havok cast member using the ‘Insert>Media Element>Havok Physics Scene’ menu option. It is very important that you establish the scale of the physics scene from the start; as we discussed earlier, scale controls the way the simulation matches real-world experience. Internally, the Havok physics simulation employs the metric system (that is, the default unit is meters). A w3d cast member may have been created in any number of world units (meters, inches, feet, user, generic). The Havok Xtra interface can work with the same units as this w3d cast member. However, in order to perform the proper simulation, Havok Xtra must know the correspondence between the display (3D scene) units and the simulation units. You must provide a world-scaling factor when initializing the physical simulation. For example, if you designed a scene using inches, then you would supply a scaling value of 0.0254 (1 inch = 0.0254 meters). Be aware that any values in the scene (like gravity, rest length of springs and so on) are interpreted as scene units rather than internal physics units. That means that a real-world gravity value of 9.81 ms^{-2} would have to be set as 386.22 in.s^{-2} if working in inches.

You must also provide a collision tolerance parameter. This tolerance is used to determine when objects are touching (that is, if they are closer than the tolerance). In general, higher collision tolerance values yield more stable simulations. However, setting too high a value could lead to

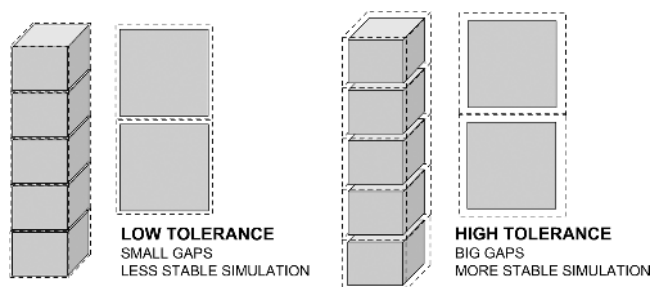


Figure 20.4 *Tolerance in a Havok simulation*

noticeable gaps between stacked objects. So it is recommended that you set the collision tolerance to the highest value at which it does not visually affect the scene. Figure 20.4 shows how tolerance affects the positioning of objects that stack.

If a scene consists of many objects in a room (crates, tables, chairs and so on) a tolerance of around 0.1 meters should be fine. However, if the objects in the scene are dice on a table a smaller tolerance, say 0.01 meters or less, is preferable. If the objects are cars or buildings, a higher tolerance applies and so on. If no value is supplied then the default tolerance of 0.1 is used. As a general rule of thumb, the tolerance value should be set to around 10% of the scaling factor used in the simulation. Collision tolerance is a value measured in scene units. That means that the scaling factor affects its actual value. ‘havok.initialize()’ must be the first Havok function called or other Havok functions will have no effect.

havok.shutdown()

This function stops the current simulation and removes it from memory.

havok.rigidBody(rbname)

This function queries the physical simulation for a rigid body of a given name. If it finds the rigid body, it returns a reference for it. You can use this to alter properties and call functions on rigid bodies.

havok.deleteRigidBody(rbname or rbindex)

This function removes a rigid body from the physics simulation given the rigid body’s name or index.

havok.makeMovableRigidBody(modelName, mass)

havok.makeMovableRigidBody(modelName, mass, isConvex)

havok.makeMovableRigidBody(modelName, mass, true, type)

This function creates a movable rigid body, with specified mass greater than zero (specified in kilograms), from a model of name ‘modelName’ and adds it to the simulation. The optional Boolean flag ‘isConvex’ indicates whether the new rigid body is to be convex or concave, where the default is convex. A model is convex when it has no hollows. Havok will create a convex version of your

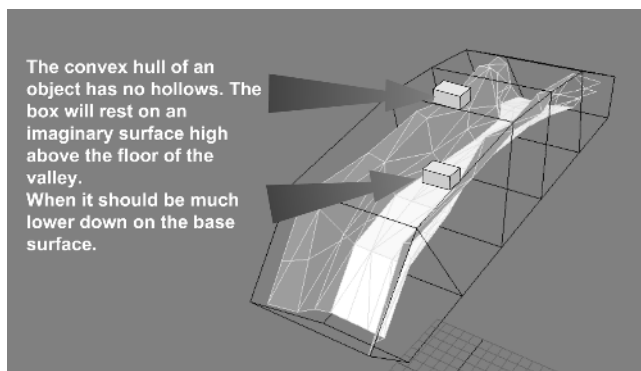


Figure 20.5 *The convex hull of a concave object*

model if you choose this option; this is called a ‘convex hull’. It is possible to use the convex hull of a model as the physical simulation in many cases. A convex hull would not work when it is important in your simulation for objects to drop into a hollow. In the later example in this chapter you will see a sledging simulation, where the slope has sides; if we used the convex hull then the sledge would sit on an imaginary surface at the top of the sides. Figure 20.5 shows the effect. Clearly this is not as intended so you would set `isConvex` to false. It is easier and faster to use convex geometries to resolve collisions, so you should use them wherever possible. When you are creating a rigid body from a model, you must add the ‘meshdeform’ modifier to the model, that is, ‘`model.addModifier(#meshDeform)`’. Otherwise, Havok Xtra cannot access the geometry of the model. The convex representation of a rigid body is called a convex hull. Experimentation is often the best way to find out how Havok handles the different options. From a performance point of view using a `#box` or `#sphere` will give the best performance, followed by using a convex hull and finally the actual model. Sometimes the best results are achieved by creating a specific proxy object to use as a convex hull. If you specify the `isConvex` parameter to be true, you can then construct a bounding sphere (`#sphere`) or axis-aligned box (`#box`) rather than the default convex hull. A sphere or box is the fastest way to handle the simulation of a rigid body.

havok.makeFixedRigidBody(modelName)

havok.makeFixedRigidBody(modelName, isConvex)

havok.makeFixedRigidBody(modelName, true, type)

This function creates a fixed rigid body from a model of name ‘modelName’ and adds it to the simulation. The optional Boolean flag ‘isConvex’ indicates whether the new rigid body is to be convex or concave (see ‘havok.makeMovableRigidBody’). The default value is convex. Fixed rigid bodies never move, but are still involved in collision detection. These are mostly used for scenery elements like walls. When you create a rigid body from a model, you must add the ‘meshdeform’ modifier to it, as in ‘`model.addModifier(#meshDeform)`’. If you don’t add the meshdeform modifier, then Havok Xtra cannot access the geometry of the model. Fixed bodies do not have mass. Mass is a property that only makes sense for objects that are free to move.

A practical example

Before you can use Havok you need to ensure that it is set up correctly. Unfortunately because of a time limit passing on the inclusion agreement, the Havok Xtra is not, at the time of writing, part of Director MX 2004, so you may have some problems in this regard. Macromedia are trying to reach an agreement with Havok to make the Xtra available to Director MX 2004 users so by the time you are reading this it may be on the CD or available as a download. Alternatively install Shockwave10 from the CD and then navigate to the 'Macromed' folder. On a Windows machine this will be in the folder 'Windows\System32\Macromed'. Inside this folder you will find a 'Shockwave10' folder that contains an 'Xtras' folder. You should find the file 'Havok.x32' in this folder. Once you have located the Xtra, you need to copy it to your Director MX 2004 installation folder. On a Windows machine this will be in 'Program Files\Macromedia\Director MX 2004\Configuration\Xtras'. Create a folder called 'Havok' inside this folder and then copy Havok.x32 to this folder. You will also need the documentation and the cast libraries, which you can get from <http://oldsite.havok.com/xtra>. Place the 'CastLibs' in a 'Havok' folder inside the 'Configuration/Libs' folder. The documentation can be placed wherever suits yourself. Once you have added these files you will need to restart Director MX 2004 before the Xtra will be available to use.

To set up the simplest Havok scene, follow the steps below. We will use Lingo to provide all the physical simulation information rather than a prepared .hke file.

- (1) First we import the 3D world; click the 'File' menu, select 'Import' and select the required Shockwave 3D (w3d) file from the dialog window. This adds the 3D cast member to the cast. This may then be dragged onto the stage.
- (2) To create the empty Havok simulation, click the 'Insert' menu, select 'Media Element>Havok Physics Scene'. This adds an empty Havok cast member to the cast.
- (3) Now add a 'Physics (No HKE)' behavior to the cast (from the 'Havok>Setup' behavior library). Playing the movie at this stage will show a static world, that is, we have not associated any rigid bodies with the 3D models in the w3d file.
- (4) Now we need to create the Havok simulation information by editing the 'Physics (No HKE)' behavior. The Lingo function below creates two rigid bodies that are added into the Havok simulation and associated with two 3D models from the w3d file: 'Box01' and 'Text01'. The first is a fixed convex rigid body, that is, one that may never move during the simulation. The second is a concave movable object.

```

1  on beginSprite
2    s = member( "HelloWorld" )
3    s.resetWorld()
4    hk = member( "Havok" )
5    hk.gravity = vector(0, -10, 0)
6    -- create ground

```

```

7   m = s.model("floor")
8   m.addModifier(#meshdeform)
9   hk.makeFixedRigidBody(m.name)
10  -- create text
11  m = s.model("hello")
12  m.addModifier(#meshdeform)
13  hk.makeMovableRigidBody(m.name, 100.0, false)
14 end

```

Listing 20.1

Listing 20.1 is from the sample ‘Examples/Chapter20/helloworld.dir’. Running the sample you will see the lettering fall to the floor surface and bounce and tumble in a very convincing manner. All this is done with just the lines of code you see in the listing. The ‘Havok Physics (No HKE)’ takes care of initializing the physics using the following parameters:

- Havok cast member 2
- tolerance 0.1
- time step 0.03
- substeps 3
- scale 1.0.

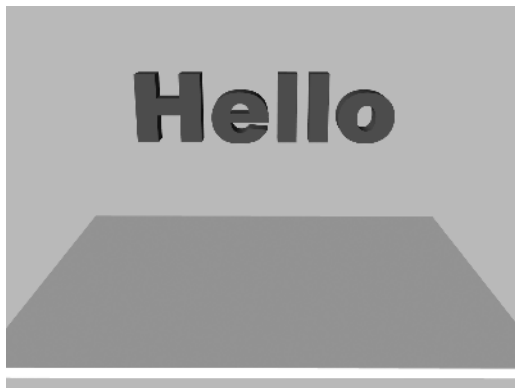


Figure 20.6 ‘Examples/Chapter20/helloworld.dir’

The scale is set to 1.0 because the geometry was created using meters as the scale. The tolerance is set to 0.1 of a meter. Try adjusting the tolerance and scale to see how it affects the scene. You could also try adjusting the gravity vector set in line 5 of Listing 20.1.

Adding keyboard control

In this second example we will build on what we have learnt so far. Open ‘Examples/Chapter20/sledgedemo.dir’. Unfortunately the geometry for this example was produced at completely the wrong size, but this illustrates how you can set up even a badly prepared 3D world to use Havok by adjusting the values for tolerance, scale and gravity. In this instance tolerance is set to 20, because at the scale of this world, that represents a small gap around the sledge. Scale is set to 0.025 and gravity to (0, -0000, 0)! Nevertheless by using these values you can create a usable simulation.

In common with many games the main work of the demo is in a main loop, which in this case is the ‘enterFrame’ handler of the 3D sprite behavior ‘3D Behavior’. Listing 20.2 shows the script. Starting with line 2 we call a simple keyboard reader. This stores whether the arrow keys and the

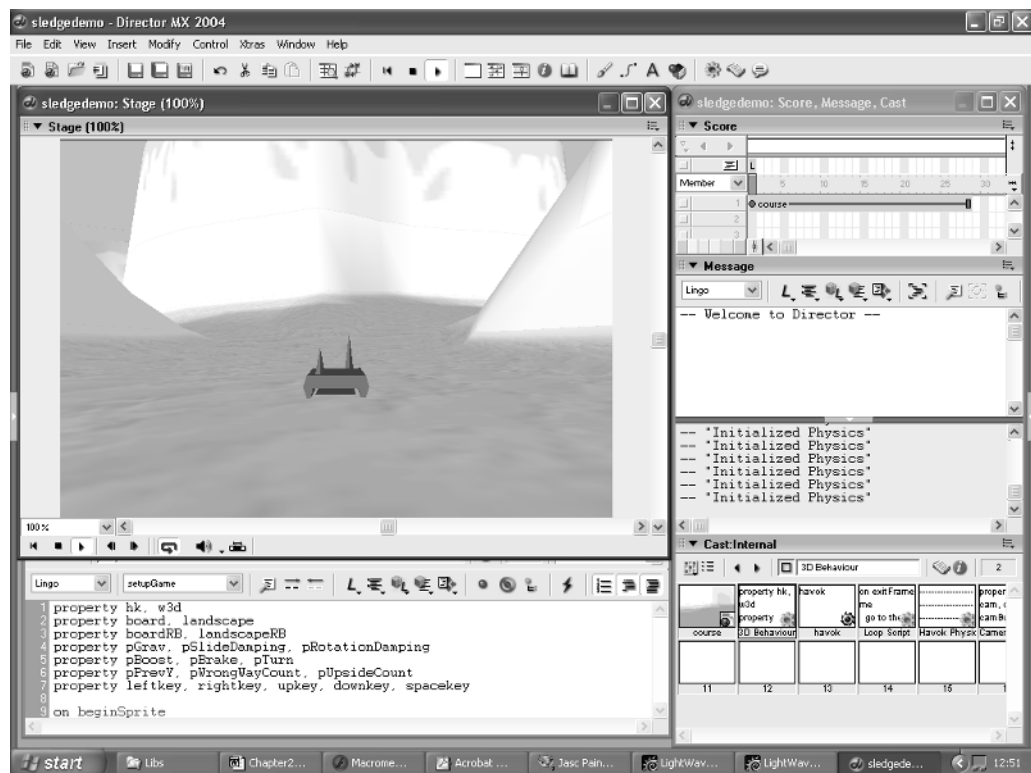


Figure 20.7 Developing 'Examples/Chapter20/sledgedemo.dir'

space key are currently pressed. The values are stored in the property variables 'leftkey', 'rightkey', 'upkey', 'downkey' and 'spacekey'. In this world there are two rigid bodies, the course and the sledge. Notice that the sledge has two strange spikes sticking up. These help handle situations where the sledge is inverted. Once the demonstration is complete we can make the sledge object invisible and replace it with a visible alternative. We did this for a popular game at www.pingu.net; the crazy sledging game uses the code below to handle most of the behavior of the sledge. The proxy object being used to control the sledges regardless of the choice of character is exactly the one shown in Figure 20.7.

A rigid body has many additional properties and methods. To find out the full details consult the Havok documentation either from your CD or from the website given earlier. One property is used in line 4 of Listing 20.3, the linear velocity; think of it as speed. Because we are going down hill the 'y' value should decrease so the difference between the previous y position and the current y position should be positive if we do the calculation shown in line 5. If it is negative then we are trying to sledge uphill and so we can use this to inform the player that they are going in the wrong direction; the code between lines 8 and 18 handles this, in this case simply by forcing the sledge to stop so the user cannot go too far in the wrong direction.

Line 21 is where we use Havok to control the motion of the sledge. Instead of simply moving the sledge we give it a nudge using the rigid body method 'applyAngularImpulse'. This takes a vector and uses the vector to apply rotational motion to the rigid body. In the example at line 21 we use the previously stored property variable 'pTurn' to give a rotation in the 'y'-axis. The left and right keys are used to control the rotation while the up and down arrows control forward and back motion. Lines 27 and 28 show how this is achieved; again we use a method of the rigid body, 'applyImpulse', which takes a vector and applies a force defined by the vector. To move the sledge we want the force for a forward motion to be along its 'z'-axis, so we first get the z-axis of the sledge, called board in line 27. This is scaled up by a boost value, which is increased or decreased in lines 14–18.

The behavior of the sledge would be highly erratic if we left it at that, so we must dampen down the effect in a separate handler, 'dampenPhysics', which we will look at in the next section.

```

1  on enterFrame
2    checkkeys
3
4    boardSpeed = boardRB.linearVelocity.magnitude
5    boardDY = pPrevY - board.worldPosition.y
6    pPrevY = board.worldPosition.y
7
8    if boardDY<0 then
9      pWrongWayCount = pWrongWayCount + 1
10   else if pWrongWayCount>0 then
11     pWrongWayCount = pWrongWayCount - 1
12   end if
13
14   if pWrongWayCount>50 and pBoost>10000 then
15     pBoost = pBoost - 200
16   else if pBoost<40000 then
17     pBoost = pBoost + 200
18   end if
19
20   if leftkey then
21     boardRB.applyAngularImpulse(vector(0,pTurn,0))
22   end if
23   if rightkey then
24     boardRB.applyAngularImpulse(vector(0,-pTurn,0))
25   end if
26   if upkey and boardSpeed<7000 and pUpsideCount = 0 then
27     hitV = board.transform.zAxis * -pBoost
28     boardRB.applyImpulse(hitV)
29   end if

```

```

30  if downkey then
31      hitV = board.transform.zAxis * pBoost
32      boardRB.applyImpulse(hitV)
33  end if
34  if not spacekey then
35      dampenPhysics
36  end if
37 end

```

Listing 20.2

Controlling the results

Physical simulations are great unless the behavior does not accurately suit your application. In this case you must try and control the behavior. One thing you cannot do is place a rigid body directly. You must use the methods of Havok to achieve the ends you desire. Some tips are shown in Listing 20.4. Lines 3–12 get the height of the ground directly below the sledge. Lines 15 and 16 show how the rotation of the sledge is reduced by applying a damping effect, which is the opposite of the sledge's angular velocity. Lines 19–28 stop the sledge sliding too much in the 'x'-axis. We want the sledge to appear as though the runners hold the snow so it is only happy when going forwards or back. The physics simulation will slide to the left and right just as easily as it will slide forward or back so we need to stop this sliding action. This should only happen when the sledge is on the ground ('boardHeight < 100'), not in the air. At line 21 we get the 'linearVelocity' of the sledge. This is a vector giving a speed in the 'x'-, 'y'- and 'z'-axes. We use the vector method 'normalize' to get a vector of unit length. Line 24 uses the vector method 'dot' to find how much of the vector is aligned with the sledge's x-axis. The result is a single value that we then map to the x-axis by multiplying the x-axis by this value in line 25. We now have a vector that expresses the direction we need to use for dampening the sideways slide. By multiplying this by the magnitude of the linear velocity and a previously stored scaling factor 'pSlideDamping', which we can use to give more or less sideways slide, we can finally, at line 27, apply this to the sledge to minimize the sideways slide.

Another thing we need to check is whether we are upside down. In any simulation this can happen. To do this we use the dot product of the sledge's 'y'-axis with the default y-axis (0, 1, 0). If this is less than 0.5 then we are currently upside down. Rather than react immediately to this we simply update a counter. This lets us adjust how quickly we reset from such a situation. In lines 42–48 you can see that we allow a count of 20 before attempting a correction. To do the correction we take a default transformation matrix, set the position of the matrix to the current sledge position then move this up a little to ensure we are above the snow. Then we use, in line 46, the 'interpolatingMoveTo' method of the rigid body. This takes two parameters, position and orientation expressed as an angle axis list. If the rigid body can be moved without interpenetrating other rigid bodies in the scene then the body is moved; if not it remains unmoved.

```

1  on dampenPhysics
2      -- check height from snow
3      boardHeight = 0

```

```

4  wpoint = board.worldPosition
5  wdown = -1.0 * board.transform.yAxis
6  idetails = w3d.modelsUnderRay(wpoint, wdown, 2, #detailed)
7  repeat with j = 1 to idetails.count
8      if idetails[j].model = landscape then
9          boardHeight = idetails[j].distance
10         exit repeat
11     end if
12 end repeat
13
14 -- reduce heading rotation
15 rbForce = boardRB.angularVelocity * -pRotationDamping
16 boardRB.applyAngularImpulse(rbForce)
17
18 -- only reduce slide when on or near the ground
19 if boardHeight<100 then
20     -- reduce sideways slide
21     rbForce = boardRB.linearVelocity
22     rbForceMag = rbForce.magnitude
23     rbForce.normalize()
24     forceXfactor = rbForce.dot(board.transform.xAxis)
25     forceXfactor = forceXfactor * board.transform.xAxis
26     xDampen = -rbForceMag * forceXfactor * pSlideDamping
27     boardRB.applyImpulse(xDampen)
28 end if
29
30 -- flip if upside down
31 tn = transform()
32 pUp = tn.yAxis
33 wUp = board.transform.yAxis
34 cosine = wUp.dot(pUp)
35
36 if cosine < 0.5 then
37     pUpsideCount = pUpsideCount + 1
38 else
39     pUpsideCount = 0
40 end if
41
42 if pUpsideCount>20 then
43     tn = transform()
44     tn.position = board.worldPosition
45     tn.position.y = tn.position.y + 50
46     boardRB.interpolatingMoveTo(tn.position, [vector(0,1,0), 0])

```

```

47     pUpsideCount = 0
48   end if
49
50 end

```

Listing 20.3

Controlling a physical simulation requires some knowledge of vector maths and you are encouraged to check out Appendix A for this detail. If this scares you then take a look at the examples that you can find on the Havok site, which give lots of ideas for controlling and dampening physics.

A real-world example

The ideas given in the sledging example are extended to controlling a snowboarder in a game we market as SnowboarderXS. In this game, which is available on the CD at 'Examples/Chapter20/SnowboarderXS.html', you choose one of two characters and courses and then slide down the course jumping ramps, collecting stars and performing tricks. The control of the snowboarder uses a proxy object but the animation you are shown of the character is controlled in code. So as you twist left the character performs a bend action. This is achieved using the code in Listing 20.4.



Figure 20.8 'Examples/Chapter/SnowboarderXS.html'

```

1 on setBoarderAnim id
2   if id<=pBoarderAnims.count And pBoarderCurAnim<>id then
3     boarder.bonesPlayer.playRate = pBoarderAnims[id].rate
4     boarder.bonesPlayer.currentTime = pBoarderAnims[id].start
5     pBoarderCurAnim = id
6   end if
7 end

```

Listing 20.4

In the ‘enterFrame’ loop, we check whether the ‘bonesplayer.currentTime’ has exceeded the value of ‘pBoarderAnims[pBoarderCurAnim].end’. If so then we set the animation back to default. Key presses that set the animation can only change the animation when the animation has been set back to default.

Summary

The Havok Xtra is a fantastic tool for creating high-quality on-line games. Hopefully Macromedia will have settled with Havok to distribute the Xtra with Director MX 2004, but if not then follow the steps given in the box earlier to give you access to world of rigid body simulations. In this chapter we covered the basics of using the control then moved on to some of the more complex ways you can control the simulations. Always remember that the most important starting point for a simulation that uses Havok is to either bring in your geometry scaled to meters or set up the scaling parameters to suit your geometry. Geometry to meter scales is the preferred route but this is not always easily achieved. You may have found some of the maths intimidating; if so, read on. Appendix A will guide you through all these dot products!

Appendix A:

Maths for games

When developing games using any development language you will find some knowledge of mathematical techniques extremely useful. In this appendix we will cover the basics that you will need to know and how this can be applied to your games in many real-world situations. All code snippets use the JavaScript syntax.

The marvel of right-angled triangles

The sprites that you will be controlling with script in your games are positioned on a coordinate system. The horizontal position is defined by the 'locH' property and the vertical position by the 'locV' property. Think of the horizontal location as the position along the 'x'-axis and the vertical location as the position along the 'y'-axis. The two axes are at right angles to each other. Consequently the positions along both the x- and y-axes can form the two sides of a right-angled triangle that meet at the right angle. This is a very important property that can be used extensively in your games. Right-angled triangles are the basis of trigonometry but before we consider that, they also have another important property.

Take a look at Figure A.1. Here we show Pythagoras' theorem. This simply states that there is a connection between the three sides of a right-angled triangle. If we know the lengths of two sides then we can determine the length of the third. The principal use you will make of this is finding the distance between two points on screen. Suppose you have a sprite at position A in Figure A.1

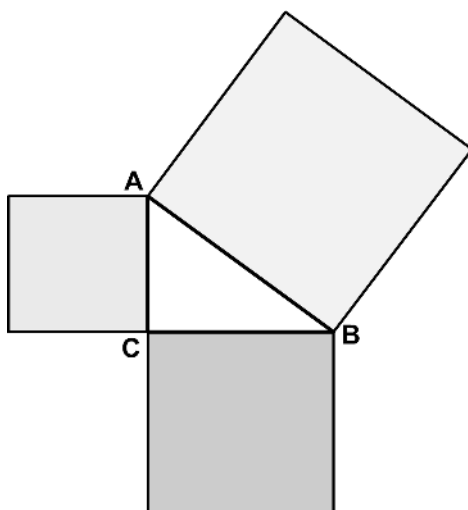


Figure A.1 *Pythagoras's theorem*

and another at position B. Then the distance from one to the other is given by the code snippet in Listing A.1.

```
1 function distanceBetween(a, b){
2     var dx = a.locH - b.locH;
3     var dy = a.locV - b.locV;
4     var dist = Math.sqrt(dx * dx + dy * dy);
5     return dist;
6 }
```

Listing A.1

We break down the problem into a distance along the 'x'-axis and a distance along the 'y'-axis. This distance is determined in lines 2 and 3. This may be a negative value, but that is not a problem because at line 4 we multiply the distance along the x- and y-axes by themselves. A number multiplied by itself is always positive because positive times positive is positive and negative times negative is positive. Then at line 4 we get the square root of the result. What we are using here is the fact that the sum of the area of the square box adjoining the side CB and the area of the box adjoining the side AC is the same as the area of the box adjoining side AB. By finding the square root of this area we discover the length of the side AB.

Finding distances between things in your games is very useful and you will often use this result. More often than not you can skip the square root calculation because this is a computationally expensive thing to do. This means it takes quite a few processor cycles to calculate. If you are concerned with the distance being under 10, for example, then this is exactly the same as the square of the distance being 100. So why determine the actual distance when you already have all the information you need?

So what is trigonometry?

Another useful feature of right-angled triangles is that they form the basis of trigonometry. You may have been less than impressed when you studied trigonometry at school, thinking that its applications were extremely limited. You could not be more wrong. Trigonometry forms the basis of the calculations in nearly all graphically dynamic games.

Figure A.2 shows a right-angled triangle. Side AB, the longest side, is called the hypotenuse or 'hyp' for short. Considering angle ABC, labeled θ in Figure A.2, makes side CB the adjacent side or 'adj' for short and side AC, the opposite side or 'opp' for short. Trigonometry is simply a method of discovering additional information about a right-angled triangle when we only know a limited amount of facts.

If the length of the side AB is 1, then the length of the side AC is described using the function 'sin' of the angle θ (see Figure A.3). Similarly the length of the side BC is given by the function 'cos'

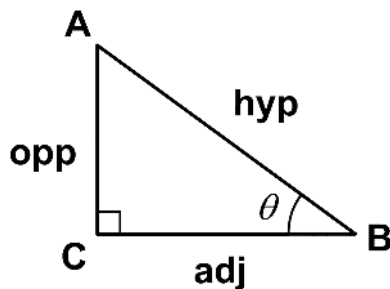


Figure A.2 *Introducing trigonometry*

of the angle θ . If the length of side AB is not 1 but 'n', then the length of side AC is given by n times sin of the angle θ and the length of side BC given by n times cos of the angle θ .

If we know the lengths of the sides then we can also use

$$\sin(\theta) = \text{opp/hyp}$$

$$\cos(\theta) = \text{adj/hyp}$$

$$\tan(\theta) = \text{opp/adj}.$$

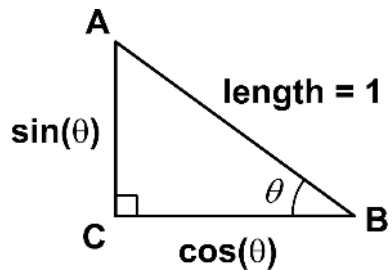


Figure A.3 Sine and cosine

So how can we use this information? In a game scenario we are very likely to know the length of AC and BC. But we are less likely to know angle θ . We know that we can calculate the length of AB using Pythagoras' theorem. To find the angle θ we use the inverse trigonometrical functions 'asin' and 'acos'. An inverse function is the reverse of the function. In a trigonometrical function you use an angle as a parameter and get a length in return. With an inverse trigonometrical function you pass a length and get returned an angle. An important thing to bear in mind is the length must be based on a hypotenuse length of 1. So you will need to scale the lengths accordingly. Listing A.2 shows how you can find the angle between two sprites.

```
1 function angleBetween(a, b){
2   var dx = a.locH - b.locH;
3   var dy = a.locV - b.locV;
4   var mag = Math.sqrt(dx * dx + dy * dy);
5   dx /= mag;
6   dy /= mag;
7   if (Math.abs(dx) > Math.abs(dy)) {
8     var theta = Math.asin(dx);
9   } else {
10    var theta = Math.acos(dx);
11  }
12  return theta;
13 }
```

Listing A.2

Notice that the function starts in an identical way by finding the distance between the two locations. This time we call the distance 'mag', short for magnitude; more about this later. Then we scale the horizontal and vertical components by this amount. The effect of this is to scale them to the size they would be if the hypotenuse was of length 1. Then we either return the result of the inverse sine function 'asin' or the inverse cosine function 'acos' depending on whether the horizontal distance or the vertical distance is greater. The passed value will be in the range -1.0 – 1.0 using the method shown.

```

1 var degtorad = 180/Math.PI;
2 for (var i=-1; i<1; i+=0.1){
3     var deg1 = Math.floor(Math.asin(i) * degtorad);
4     var deg2 = Math.floor(Math.acos(i) * degtorad);
5     trace(i + " ASin: " + deg1 + " ACos:" + deg2);
6 }
7 stop();

```

//Output from above listing

```

-1.0 ASin: -90 ACos: 180
-0.9 ASin: -65 ACos: 154
-0.8 ASin: -54 ACos: 143
-0.7 ASin: -45 ACos: 134
-0.6 ASin: -37 ACos: 126
-0.5 ASin: -31 ACos: 120
-0.4 ASin: -24 ACos: 113
-0.3 ASin: -18 ACos: 107
-0.2 ASin: -12 ACos: 101
-0.1 ASin:  -6 ACos:  95
 0.0 ASin:  -1 ACos:  90
 0.1 ASin:   5 ACos:  84
 0.2 ASin:  11 ACos:  78
 0.3 ASin:  17 ACos:  72
 0.4 ASin:  23 ACos:  66
 0.5 ASin:  29 ACos:  60
 0.6 ASin:  36 ACos:  53
 0.7 ASin:  44 ACos:  45
 0.8 ASin:  53 ACos:  36
 0.9 ASin:  64 ACos:  25
 1.0 ASin:  89 ACos:   0

```

Listing A.3

A function can only return a single value based on the parameter passed. A sine or cosine graph is shown in Figure A.4. You can see that the function sine is zero when the angle is either 0 or 180°. This creates a potential problem. For this reason the return values from the 'asin' function will be in the range -90 to +90° and the return values from the 'acos' function in the range zero to 180°. To allow for a full rotation you will need to check for other features of the locations.

Looking at Figure A.5 you can see that when Sprite b has a 'locV' value that is greater than that of Sprite a, the value for θ will be less than 180° so we can take the value return from 'acos' directly to determine the angle. If, however, the position of Sprite b is above that of Sprite a on screen then the value for θ will be greater than 180°. Therefore we need to modify the result by subtracting the return value from acos from 360°.

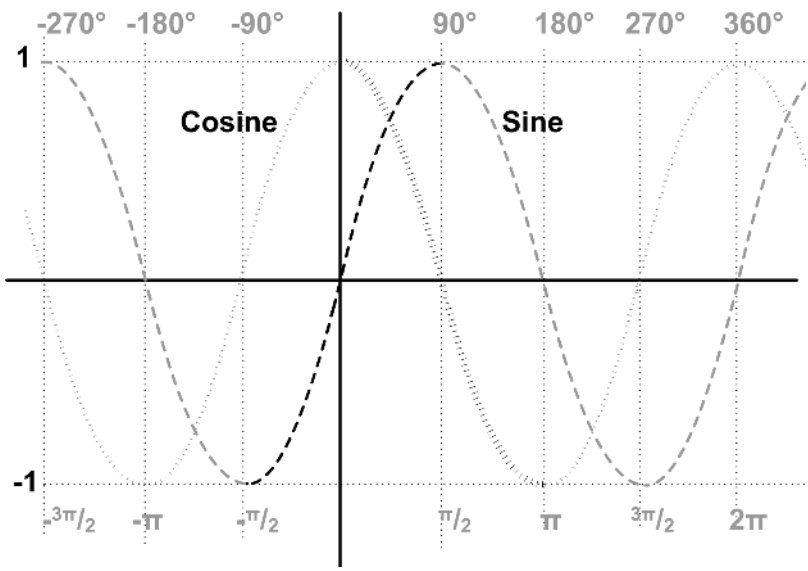


Figure A.4 Graphs of the sine and cosine functions

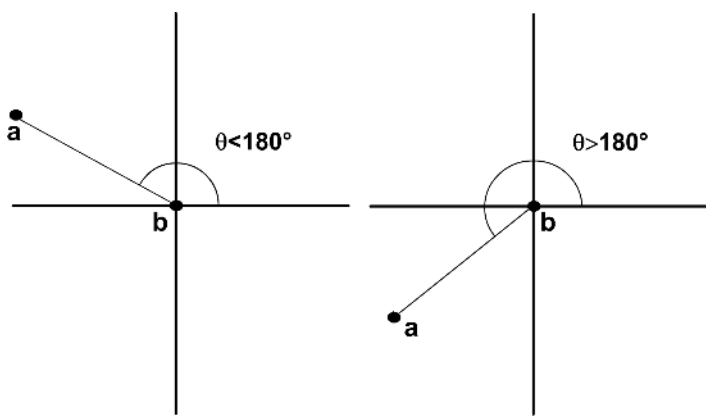


Figure A.5 θ less than and greater than 180°

So far we have been talking in terms of degrees. But you will find that all the trigonometrical functions use radians instead of degrees. When you think of degrees you imagine a circle divided into 360 segments. Radians divide a circle up in a similar way but this time they use two times [π] as the total. Essentially degrees and radians are the same thing, a division of a circle into segments, but the scale of the actual numbers differs; there are approximately 57 times as many degrees as radians in a full rotation. To convert between degrees and radians you simply need to multiply or divide by a constant value:

$$\begin{aligned} \text{degree} &= \text{radian} * 57.29578 \\ \text{radian} &= \text{degree}/57.29578. \end{aligned}$$

The value 57.29578 is $180/[\pi]$. Hence the first line of Listing A.3.

```

1  var pRed, pBlue;
2
3  function beginSprite(){
4      pRed = sprite(this.spriteNum);
5      pBlue = sprite(1);
6  }
7
8  function enterFrame(){
9      pRed.locV = _mouse.mouseV;
10     pRed.locH = _mouse.mouseH;
11     angleBetween(pRed, pBlue);
12 }
13
14 function angleBetween(a, b){
15     var dx = a.locH - b.locH;
16     var dy = a.locV - b.locV;
17     var mag = Math.sqrt(dx * dx + dy * dy);
18     dx /= mag;
19     dy /= mag;
20     var theta = 0;
21     var degtorad = 180/Math.PI;
22     var segment;
23
24     if (Math.abs(dx)>Math.abs(dy)) {
25         theta = Math.asin(dx);
26         if (dy<0){
27             segment = "sin1 ";
28             theta = Math.PI/2 - theta;
29         }else{
30             segment = "sin2 ";
31             theta += Math.PI * 1.5;
32         }
33     }else{
34         if (dy<0){
35             segment = "cos1 ";
36             theta = Math.acos(dx);
37         }else{
38             segment = "cos2 ";
39             theta = Math.PI * 2 - Math.acos(dx);
40         }
41     }

```

```

42  member("angle").text = segment + Math.floor(theta * detorad);
43  }

```

Listing A.4

Listing A.4 shows the behavior script from 'Examples/AppendixA/inversetrig.dir'. The example allows the user to move a sprite. The angle between the sprite and a second sprite in the center of the screen is calculated for each screen update. The function 'angleBetween' is called to do the calculation. Most of the function has been discussed previously. The exception is handling the return value from the 'asin' function. Here we need to do some adjusting to get a return value that gives a smooth value throughout a full rotation. We juggle the value based on the on-screen position of one sprite in relation to another. If the sprite that moves with the mouse is positioned above the other then we take the value returned from asin from half $[\pi]$, if it is below the other then we add $[\pi]$ times 1.5 to the return value.

Introducing vectors

So now we are able to determine an arbitrary angle based on screen locations. Let us use this ability to create a pool game. We want to be able to bounce a ball off another. The problem is summarized in Figure A.6

If a ball is moving from one frame to the next using an increment for the horizontal location and an increment for the vertical location then we are using vectors without even knowing. A vector is a direction and magnitude in any dimensional space. For your games you will use either 2D vectors, for 2D games or 3D vectors, which are built into Director for the 3D Xtra. The first thing we need to calculate is whether we are going to hit another ball. Because the balls will all have the same radius this is another example of using Pythagoras. If the radius of both balls is 10 and the square of the distance between them is 200 they are touching. So we need to calculate the bounce.

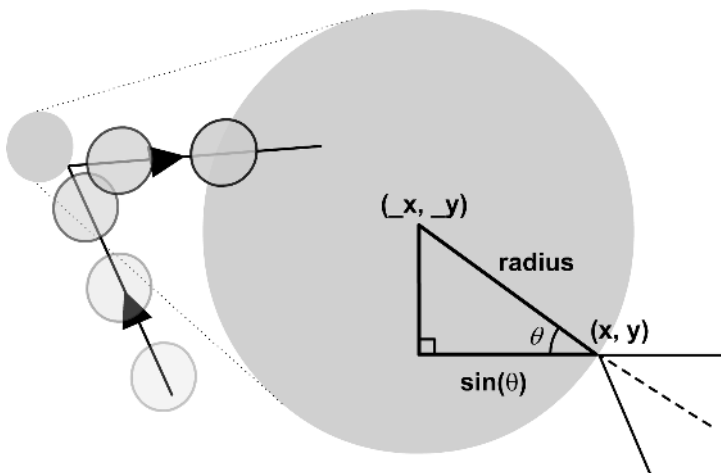


Figure A.6 Bouncing pool ball

If one ball is fixed then the problem is made easier. In this case we can calculate the angle of the dotted line in Figure A.6 using the 'angleBetween' function that is listed above. We can calculate our current angle using the values we have for the movement in the horizontal and vertical directions. Now we have two angles. We need to flip our current angle around the dotted line which we can easily do. First we find the difference between the two angles. If the value is positive then we subtract the difference from the angle of the dotted line and if it is negative then we add the value. At this stage we have an angle; now we can calculate new values for the movement components using this new angle.

```

1  var pRed, pBlue, pVec, pBounce;
2
3  function beginSprite(){
4      pRed = sprite(this.spriteNum);
5      pBlue = sprite(1);
6      pVec = new Object();
7      startRoll();
8  }
9
10 function enterFrame(){
11     pRed.locH += pVec.dx;
12     pRed.locV += pVec.dy;
13     if (distanceBetween(pRed, pBlue)<16 && !pBounce){
14         bounce(pRed, pBlue);
15     }else if (pRed.locH<-10 || pRed.locH>350
16             || pRed.locV<-10 || pRed.locV>250){
17         startRoll();
18     }
19 }
20
21 function bounce(){
22     var theta = angleBetween(pRed, pBlue);
23     var curdir = angleBetween(point(0, 0), point(pVec.dx, pVec.dy));
24     var diff = theta - curdir;
25     var theta1 = curdir + diff + Math.PI;
26     var mag = Math.sqrt(pVec.dx * pVec.dx + pVec.dy * pVec.dy);
27     pVec.dx = Math.cos(theta1) * mag;
28     pVec.dy = Math.sin(theta1) * mag;
29     pBounce = true;
30 }
31
32 function startRoll(){
33     pRed.locV = Math.random() * 300;

```

```

34  pRed.locH = Math.random() * 300;
35  pVec.dx = (pBlue.locH - pRed.locH)/20.0;
36  pVec.dy = (pBlue.locV - pRed.locV)/20.0;
37  pBounce = false;
38 }
39
40 function distanceBetween(a, b){
41   var dx = a.locH - b.locH;
42   var dy = a.locV - b.locV;
43   return Math.sqrt(dx * dx + dy * dy);
44 }
45
46 function angleBetween(a, b){
47   var dx = a.locH - b.locH;
48   var dy = a.locV - b.locV;
49   var mag = Math.sqrt(dx * dx + dy * dy);
50   dx /= mag;
51   dy /= mag;
52   var theta = 0;
53
54   if (Math.abs(dx)>Math.abs(dy)){
55     theta = Math.asin(dx);
56     if (dy<0){
57       theta = Math.PI/2 - theta;
58     }else{
59       theta += Math.PI * 1.5;
60     }
61   }else{
62     if (dy<0){
63       theta = Math.acos(dx);
64     }else{
65       theta = Math.PI * 2 - Math.acos(dx);
66     }
67   }
68
69   return theta;
70 }

```

Listing A.5

Listing A.5 is from 'Examples/AppendixA/pool.dir'. It extends the ideas presented so far. We have a ball rolling towards another ball. We want the ball to bounce off if the distance between the two balls is under 16 (see line 13). If this is the case then the function 'bounce' is called. Here we get the angle between the two balls and the angle of the ball's direction. Then we calculate the difference

between the two angles. We now have the information to set the new direction after the bounce. Line 25 shows how the current direction is used along with the difference between the current direction and the vector between the two balls. Finally the resulting angle is flipped by adding $[\pi]$ to the result. $[\pi]$ is the radian equivalent to 180° or a reversal in direction. This gives a new angle after the bounce; now we need to use this angle to determine new movement amounts in the vertical and horizontal directions. We get the current magnitude of this movement vector by a Pythagoras calculation at line 26. Then we use the functions 'cos' and 'sin' to apply this to the property vector 'pVec'.

Moving into 3D

The knowledge we have gained so far can be extended into the third dimension. In 3D space, points have three components and so too have vectors. A vector expresses the movement along the 'x', 'y' and 'z'-axes. An example of using vectors is shown in Listing A.6.

```
1 function startMovie(){
2   var v1 = vector(10, 20, 30);
3   var v2 = vector(45, 55, 65);
4   trace ("The magnitude of v1 is " + v1.magnitude);
5   trace ("The angle between v1 and v2 is " + ↵
        v1.angleBetween(v2));
6 }
```

Listing A.6

3D vector methods

A vector object has several useful methods.

normalize

`vec.normalize()`; converts the vector to unit length. A vector of unit length is useful when doing calculations that involve trigonometrical functions.

dot

```
c = v1.dot(v2);
```

The dot product of two vectors is defined as

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta, \quad \text{where } 0 \leq \theta \leq 180^\circ.$$

The symbol $|\mathbf{a}|$ refers to the magnitude of the vector \mathbf{a} . The dot product is a scalar; this simply means it is a number with a single component. Given two vectors $\mathbf{a} = [a_x, a_y, a_z]$ and $\mathbf{b} = [b_x, b_y, b_z]$,

the dot product is given by

$$\mathbf{a} \cdot \mathbf{b} = a_x \times b_x + a_y \times b_y + a_z \times b_z.$$

The dot product is very useful for finding angles between vectors. Because we know that

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta,$$

this implies that

$$(\mathbf{a} \cdot \mathbf{b}) / (|\mathbf{a}| |\mathbf{b}|) = \cos \theta.$$

cross

```
v3 = v1.cross(v2);
```

This method creates a further vector that is at right angles or *orthogonal* to the two vectors used in the cross product. Unlike the dot product the operation is not commutative. This simply means that $\mathbf{A} \times \mathbf{B}$ does not necessarily equal $\mathbf{B} \times \mathbf{A}$, whereas $\mathbf{A} \cdot \mathbf{B}$ equals $\mathbf{B} \cdot \mathbf{A}$. The cross product of two 3D vectors is given by

$$\mathbf{A} \times \mathbf{B} = [A \cdot y \cdot B \cdot z - A \cdot z \cdot B \cdot y, A \cdot z \cdot B \cdot x - A \cdot x \cdot B \cdot z, A \cdot x \cdot B \cdot y - A \cdot y \cdot B \cdot x].$$

angleBetween

```
theta = v1.angleBetween(v2);
```

returns the angle between two vectors in degrees.

distanceTo

```
dist = v1.distanceTo(v2);
```

returns the distance between two vectors. It is more like using the vector to express a location rather than a direction.

Matrices

The final link in your mathematical arsenal will be the matrix. This is simply a block of numbers. In the 3D world of Director MX 2004 a matrix is called a transform. This is simply a specialist type of matrix. It contains 16 values stored in a block arrangement. Before we look at 4×4 matrices, which are handled by Director's internal methods, we will look at a simple 3×3 matrix:

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

By combining vectors with the transformation matrix the scene is manipulated. If a vector exists $[x, y, z]$ then the result of combining with the matrix above is given by

$$X = a*x + b*y + c*z;$$

$$Y = d*x + e*y + f*z;$$

$$Z = g*x + h*y + i*z.$$

The column vector is multiplied by each row of the matrix.

Vector and matrix operations

Thankfully the matrix calculations can all be performed using methods built into both the vector and matrix objects.

transform

`t = transform()`; creates an identity transform. An identity transform is one that will have no effect on the vectors that are multiplied by it or on another matrix that is multiplied by it.

position, scale, rotation

`v = t.scale`; returns the position, scale and rotation of a transform. You can also use the property to set the transform but this is not recommended.

xaxis, yaxis, zaxis

`v = t.xaxis`; returns the axes of the transform in a vector object.

axisAngle

`aa = t.axisAngle`; returns a list that includes the rotation axis and the angle of rotation about this axis.

rotate

`t.rotate(x, y, z)`;

`t.rotate(point, vector, angle)`;

The transform is rotated by the amounts specified for each axis or using the angle axis specified.

preRotate

`t.preRotate(x, y, z)`;

`t.preRotate(point, vector, angle)`;

The transform is rotated by the amounts specified for each axis or using the angle axis specified before the current transformation.

translate

```
t.translate(x, y, z);
```

The transform is translated by the amounts specified for each axis.

preTranslate

```
t.preTranslate(x, y, z);
```

The transform is translated by the amounts specified for each axis before the current transformation.

multiply

```
t1.multiply(t2);
```

alters the original by multiplying by the parameter transform.

preMultiply

```
t1.preMultiply(t2);
```

alters the original by multiplying by the parameter transform before the current transformation.

interpolate

```
t3 = t1.interpolate(t2, percent);
```

returns a new transformation matrix, which is 100 percent the original and percent the passed parameter.

interpolateTo

```
t1.interpolate(t2, percent);
```

adjusts the original transform so it is 100 percent the original and percent the passed parameter.

identity

```
t.identity
```

resets the transform to the identity matrix.

invert

```
t.invert()
```

turns the transform into the inverse of itself.

inverse

```
t2 = t1.invert()
```

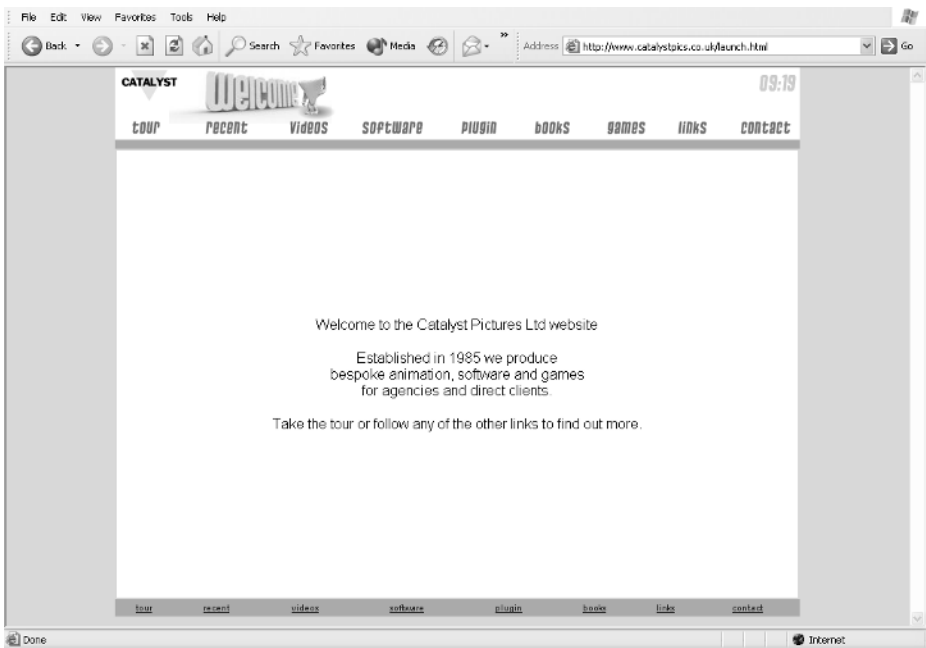
returns a transform that is the inverse of the original. If t2 is multiplied by t1 then an identity matrix is the result.

Summary

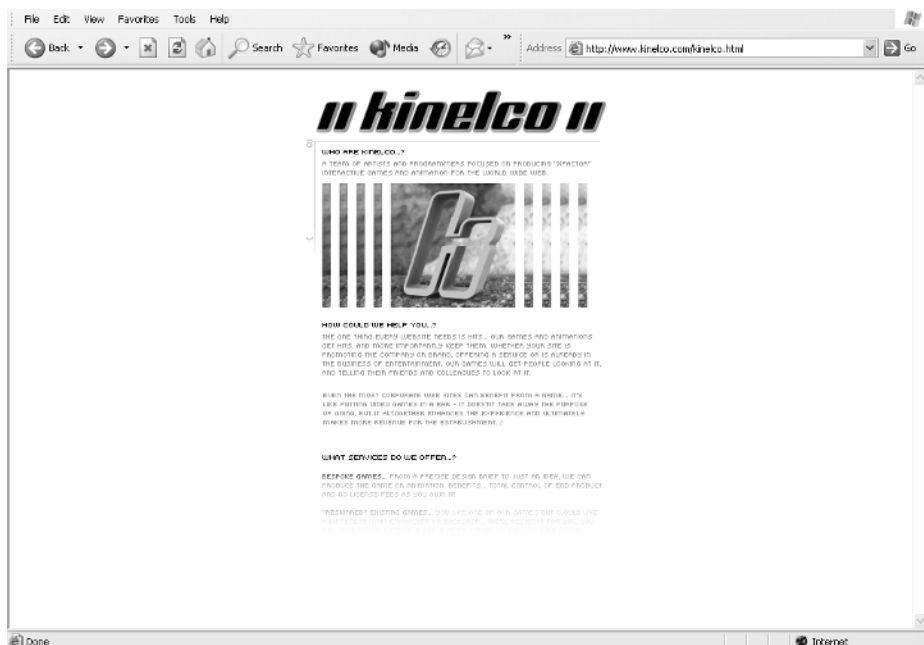
Hopefully this appendix will have introduced some of the more important maths techniques that you will use in your games. The topic is the subject of some good books, but unfortunately most of these tend to be too deep and require a graduate level understanding of mathematics. The web is a useful source of additional information.

Appendix B:

Links

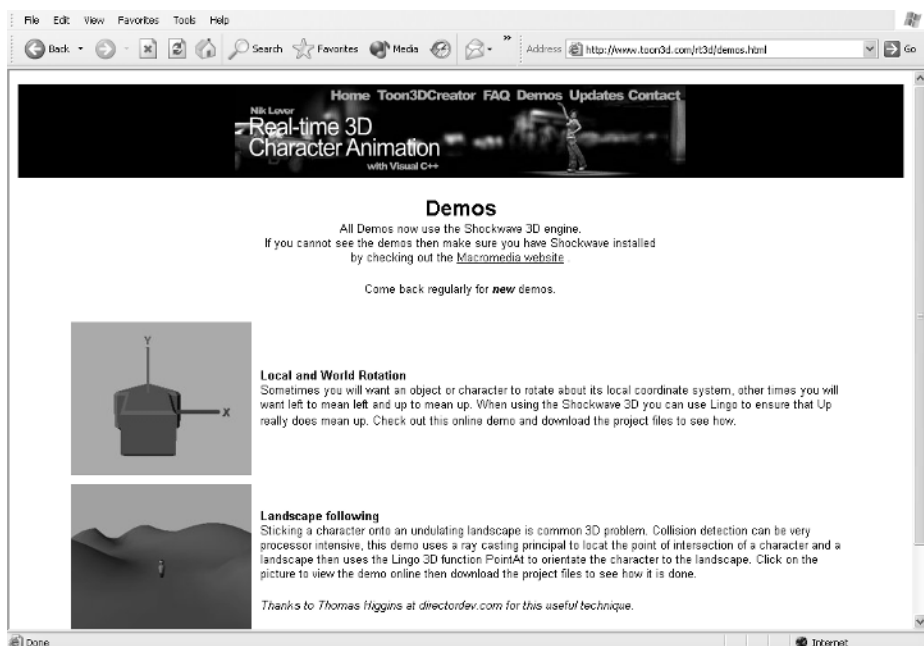


www.catalystpics.co.uk
Home page of the author’s company.



mohsye.com

After-hours site created by Catalyst staff Christian Holland and Paul Barnes.



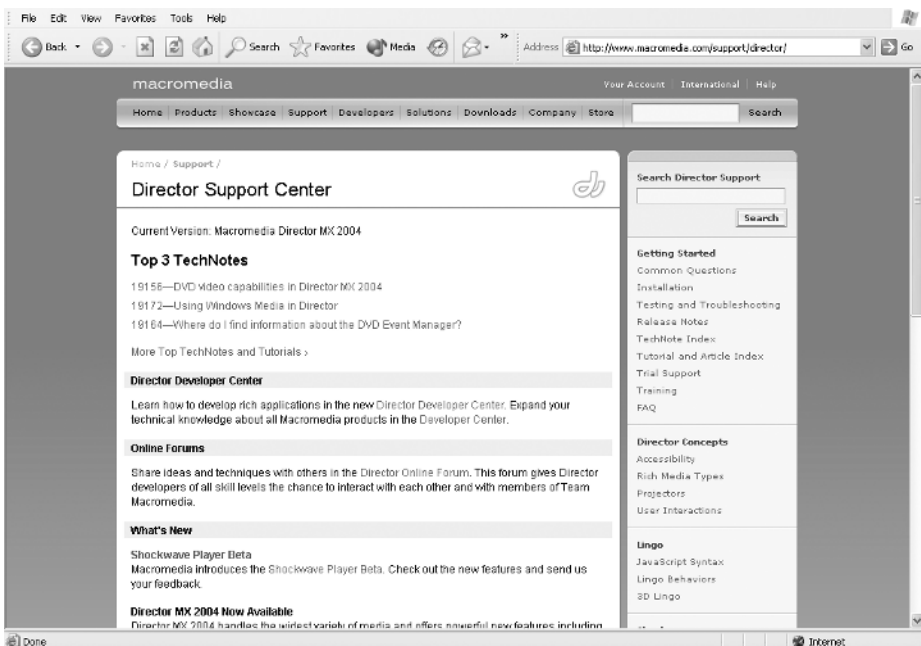
www.niklever.net/director

Home page for the book; check it out for useful bug fixes and links.



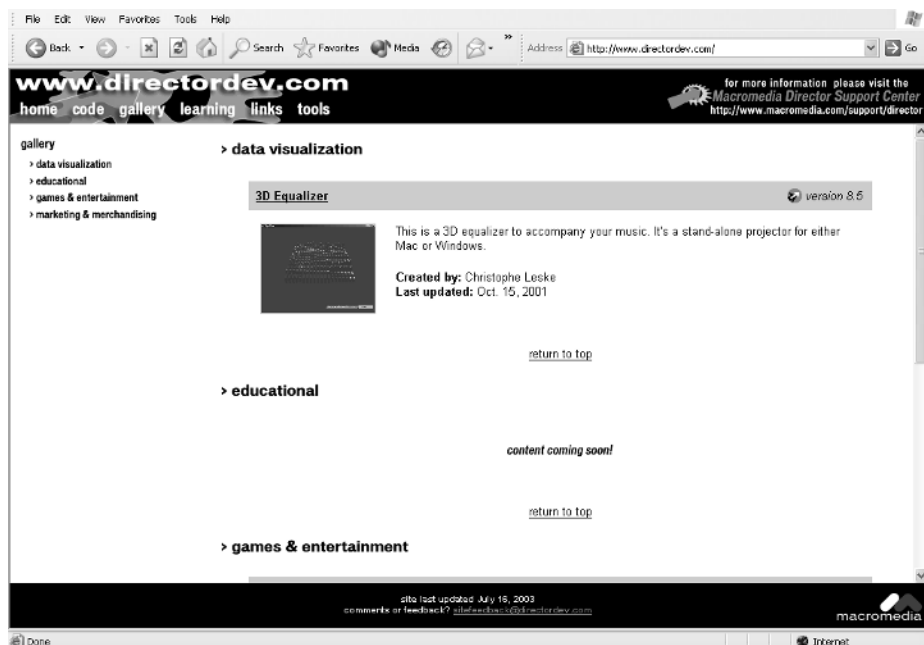
www.frosties.co.uk

We have produced several Shockwave 3D games for this site. Take a look at them.



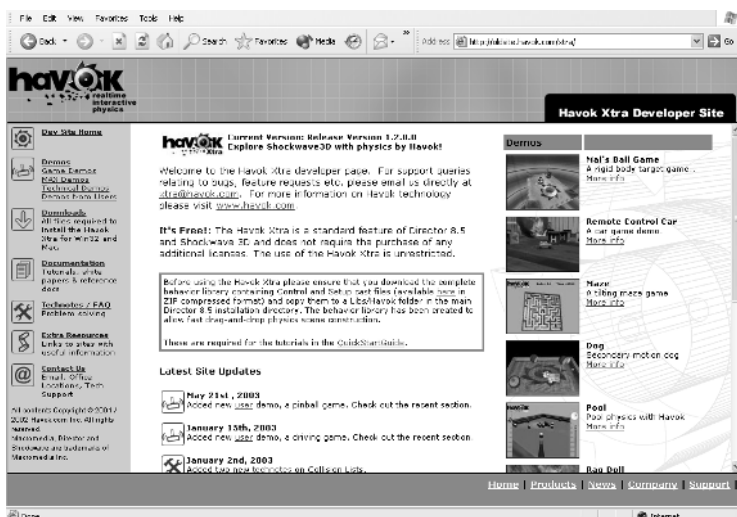
http://www.macromedia.com/software/director/?promoid=home_prod_dir_082403

The home page for Director on Macromedia's excellent site.



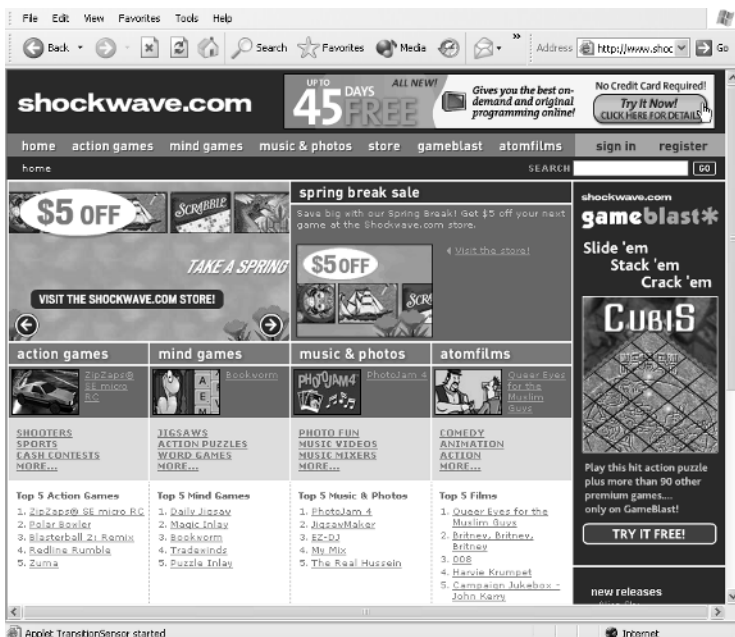
www.directordev.com

Full of code snippets to help you with your development.



http://oldsite.havok.com/xtra

Source of documentation and samples for the Havok Xtra.



Shockwave.com

Source of some of the best on-line games around.



miniclip.com

A popular game site, which we regularly provide with games.

The following is a list of some other useful Director web sites.

Director Online (www.director-online.com)

DOUG contains articles, downloadable files and searchable archives.

MediaMacros (www.mediamacros.com)

This site has downloads, games, articles, forums and more.

DirectorWeb (www.mcli.dist.maricopa.edu/director/)

On the web since August 1994 (Dir 3.x!), DirWeb is for the vast legions of Macromedia Director users/abusers/wannabes/unflashable, die-hard zealots of the ultimate multimedia authoring tool in this or any other universe. Content is free, dynamic, bannerless and subtly opinionated.

Update Stage (www.updatestage.com)

This site is a great resource for third party Xtras and also has in-depth articles.

DirectXtras Web Site (www.directxtras.com)

DirectXtras creates Xtras for Macromedia Director, Authorware and Shockwave, which empower developers to deliver content to end users via the most unique, user-friendly methods.

Integration New Media (www.integrationnewmedia.com/)

As one of the main Xtra developers for Director and Authorware, Integration New Media (INM) offers free fully functional trial versions of all their products, along with tutorials, demos and sample code. Also on the site are technical notes and white papers as well as a gallery, which showcases projects created with INM products.

Zeus Productions (www.zeusprod.com)

This site includes downloads and articles.

FBE Director Online Learning (www.fbe.unsw.edu.au/Learning/Director/)

Contains tutorials and examples of Director content.

Shocknet (www.shocknet.org.uk/index.asp)

'If you want to work with netLingo and find out how to interface with a web based database then this is the place for you.'

ShockSites (www.shocksites.com)

This site has been developed to help people find Director web sites, resources and Shockwave examples.

Xtrasy (www.xtrasy.com)

A valuable resource for Macromedia Director Xtras developers. A complete list of Xtras currently available for Director, a list of useful books for developers and a collection of useful links related to our favorite software.

Lingo Workshop (www.lingoworkshop.com)

Mecca Medialight's Lingo Workshop site has Shockwave demos (many with source code), some articles, Director utilities and the occasional tutorial.

Director-3D.com (www.director-3d.com)

Tutorials, demos and games using Director Shockwave 3D.

The Burrow (www.theburrow.com)

This site has 3D demos and tutorials.

Skeleton Moon (www.skeletonmoon.com)

Site with sample code and utilities for Director with an emphasis on 3D in Director.

ForgeFX (www.forgefx.com)

ForgeFX is a full service software development studio that specializes in developing Shockwave 3D content. Features examples of content built with Director.

Dave Mennenoh's Director Pages (www.blurredistinction.com/director/)

From the author of *Director 8.5: the Complete Reference*. This site includes Director demos, utilities and tips. It also has links to other Director resources.

SetPixel (www.setpixel.com)

SetPixel is an inspirational resource/playground subsidized by SetPixel Corporation (<http://corp.setpixel.com/>). Its purpose is to release enlightening articles, demos and source based on strong concepts rather than the loose, general application of technology.

MultimediaHelp.org (www.multimediahelp.org/home/director.html?action=downloads)

Free Director source movies and utilities.

James Newton's Director Behavior Demonstrations (perso.planetb.fr/newton/)

This site includes downloadable behaviors for use in Director.

Jim Collins Technotes (venuemedia.com/mediaband/collins/technotes.html)

This site has in-depth articles and lots of Shockwave examples in the Smoke and Mirrors section.

Teton Multimedia (www.tetonmultimedia.com/tetonmultimedia/lessons.cfm)

Director articles and source code.

Quantum Wave Interactive Inc. (www.quantumwave.com)

This site has demos and downloadable Director content.

[#Lingo: Programmer.com] (www.lingoprogrammer.com)

A site created by Stuart Gerstein, dedicated to the advancement of knowledge, logic and object-orientated programming systems.

Ultimate 3D Links Shockwave 3D Community (www.3dlinks.com/community_shockwave3D.cfm)

3DLinks.com is one of the largest information portals targeted specifically at the 3D creative developer market.

Shockwave Multiuser Pages (poppy.macromedia.com/multiuser/)

This site provides a place for the community of Shockwave multi-user developers to chat and share source code.

Tabuleiro (xtras.tabuleiro.com)

Company that specializes in Director Xtras including the Nebulae multi-user server.

Developer Dispatch (www.developerdispatch.com)

Developer Dispatch is a web log (blog) published by Gary Rosenzweig. This site contains information for Flash and Director developers.

Director French Forum (director.media-box.net)

Director forums in French.

Bibliography

Art

The Illusion of Life (Frank Thomas, Ollie Johnston). ISBN 0 89659 232 4, Abbeville 1981. Written by two of the nine old men of Disney, this book contains a wealth of information for anyone interested in animation.

The Animator's Survival Kit (Richard Williams). ISBN 0 57120 228 4, Faber and Faber 2001. The best practical guide to creating great animation ever written, by one of the best in the business.

Game Art: The Graphic Art of Computer Games (Leo Hartas, Dave Morris). ISBN 0 82302 080 0, Watson-Guption Publications 2003. More than just an art book, the volume is full of interviews with game artists, programmers and other computer luminaries. Discussions outline the process that game design teams go through when choosing the point of view of the player character, interiors and the 'feel' of the games, from proto-mythological to urban noir and the visual choices that govern the experience of the gamer.

Ultimate Game Design: Building Game Worlds (Tom Meigs). ISBN 0 07222 899 7, McGraw-Hill Osborne Media 2003. Build games with techniques and insights from a pro. Author and game developer Tom Meigs shows you the finer points of world building and behavior scripting. Learn about level stubbing, lighting, prop and item placement, camera tricks, particle and effects systems, communication loops, massively multi-player on-line games and much more.

Scripting

Code Complete: A Practical Handbook of Software Construction (Steve C. McConnell). ISBN 1 55615 484 4, Microsoft Press 1993. A modern-day classic on software engineering, this book focuses on specific practices you can use to improve your code and your ability to debug it and ultimately deliver better, more efficient programs in less time.

Lingo in a Nutshell (Bruce A. Epstein). ISBN 1 56592 493 2, O'Reilly & Associates 1998. This book has to do with behind-the-scenes aspects of Lingo, including file management, data structures, loops, conditionals and event handlers. Simply put, it treats Lingo as a programming language, rather than merely as a piece of Macromedia Director.

Advanced Lingo for Games (Gary Rosenzweig). ISBN 0 78972 331 X (book and CD-ROM edition), Que 2000. Teaches you to use advanced Lingo, the programming language behind Macromedia Director, by creating games. Features 21 games plus variations.

JavaScript: The Definitive Guide (David Flanagan). ISBN 0 59600 048 0 (4th edition), O'Reilly & Associates 2001. JavaScript is now part of Director. Flanagan's work is an excellent book for programmers interested in learning it quickly.

Flash

Action Script for Flash MX (Colin Mook). ISBN 0 59600396 X (2nd edition), O'Reilly & Associates 2003. Definitive action script bible.

Flash MX 2004 Games (Nik Lever). ISBN 0 24051963 9, Focal Press 2004. Companion volume to this book.

Active Server Pages (ASP) development

SAMSTeach Yourself ASP.NET in 24 Hours Complete Starter Kit (Scott Mitchell). ISBN 0 67232 543 8 (book and CD-ROM edition), SAMS 2003. This book teaches nonprogrammers how to utilize the powerful built-in ASP.NET controls while using the free web authoring tools.

Macromedia Dreamweaver MX 2004 with ASP, ColdFusion, and PHP: Training from the Source (Jeffrey Bardzell). ISBN 0 32124 157 6 (book and CD-ROM edition), Macromedia Press 2003. Text shows how to use Dreamweaver's built-in server behaviors and application objects to assist in the rapid development of dynamic web applications.

3D

3D Math Primer for Graphics and Game Development (Fletcher Dunn, Ian Parberry). ISBN 1 55622 911 9 (1st edition), Wordware Publishing 2002. Covers fundamental 3D maths concepts that are especially useful for computer game developers and programmers. Illustrates how to put the techniques into practice and exercises at the end of each chapter help reinforce the concepts.

Tricks of the 3D Game Programming Gurus – Advanced 3D Graphics and Rasterization (André LaMothe). ISBN 0 67231 835 0 (book and CD-ROM edition), SAMS 2003. This book helps its readers make great progress in creating 3D worlds and the action that goes on in them.

Physics for Game Developers (David M. Bourg). ISBN 0 59600 006 5 (1st edition), O'Reilly & Associates 2001. This book reviews the entire maths for creating realistic motion and collisions for cars, aeroplanes, boats, projectiles and other objects along with C/C++ code for Windows.

Shockwave 3D (Jason Wolf). ISBN 0 73571 197 6 (book and CD-ROM edition), New Riders 2002. Explains how to use Shockwave 3D to add the third dimension to movies created in Director.

Macromedia Director 8.5 Shockwave Studio for 3D: Training from the Source (Phil Gross, Mike Gross). ISBN 0 20174 164 4 (book and CD-ROM edition), Macromedia Press 2001. Shows how to use Intel Internet 3D graphics software that ships with Director. Helping developers deliver interactive 3D content over the web.

Index

- 3D application:
 - angles and axis of rotation, 245–6
 - describing, 3D space, 241–2
 - describing an object, 243–4
 - with Director, 241
 - Euler angles, 244–5, 274–5
 - ‘hither and yon’ settings, 249
 - matrices, 312–15
 - ‘origin’, 241
 - polygon normals, 244
 - rotating a box, 244
 - rotation not about the origin, 246–7
 - scaling the object, 247–9
 - transformations, 242–3
 - vectors, 242, 311–12, 314–15
 - ‘z’-buffers, 249, 256–7
 - see also* w3d type files for Director
- 3D techniques:
 - about 3D techniques, 274
 - aligning to a terrain, 284–7
 - camera moving, 274–80
 - collision testing, ‘modelsUnderRay’ method, 281–4
 - tracking with a camera, 280–1
- Absolute value (abs), 81
- Abstract imagery, 50–1
- Adobe Photoshop, use of, 6
- ai file format, 30
- Aligning to a terrain, 284–7
- Animation:
 - about animation, 1
 - animation modifier for w3d files, 272
 - see also* Computer generated imagery (CGI), for animation; Walk cycle (with Flash MX 2004)
- Anti-aliasing, 8–9, 40
- Arrays and lists:
 - with board games, 166
 - with Flash, 145
 - JavaScript arrays, 83–5
 - Lingo lists, 83–4
- Artwork creation with Director, 1–2
- ASP for database connections, 186–9
- Background creation, 46–55
 - abstract imagery, 50–1
 - complex backgrounds, 48–9
 - from computer graphic packages, 51–3
 - giving characters space, 49–50
 - gradients with complex art, 51–2
 - importing from other sources, 50
 - keep designs simple, 46–7
 - keylines, 47–50
 - ‘Moon and stars’ by Suzie Webb, 47
 - with photography, 55
 - ‘Simple bedroom’ by Suzie Webb, 47–8
 - tiled backgrounds, 53–4
- Ball creation, 2–9
- Binary numbers, 24
- Bitmaps:
 - about bitmaps, 3, 23–5
 - naming, 3–4
 - and vector graphics, 25
- Bits, 23–4
- bmp file format, 53
- Board games, 164–81
 - basic method, 165–6
 - best move evaluation, 174–9
 - board adjusting, 172–3
 - board checking, 169–71
 - board scanning, 171–2
 - building the board, 166
 - ‘computerScore’ function, 175
 - improving the evaluation, 179–80
 - initializing the board, 167–9
 - ‘legalMove’ function, 175–8
 - player’s move tracking, 169–73
 - ‘playerScore’ function, 179
 - presentation, 180–1
 - Reversi, 164–5
 - two-player games, 164
- Bounding box, Flash MX 2004, 35
- Bytes, about bytes, 7, 24
- Camera moving, 274–7
 - following an object (tracking), 280–1

- Camera moving (*Continued*)
 - improving the motion, 277–80
 - see also* Euler angles
- Cameras in w3d files, 263, 265–6
 - properties, 267–9
- Cast window, 3
- Catalyst Pictures (animation creation company), 57
- CGI *see* Computer generated imagery (CGI), for animation
- Character design, 30–1
 - dramatic performance capabilities, 31
 - golden rules, 30
 - segmenting a character, 34–6
 - see also* Walk cycle
- Charlie *see* Modeling Charlie (female character)
- Collision modifier for w3d files, 272
- Collision testing:
 - 3D ‘modelsUnderRay’ method, 281–4
 - platform games, 203
- Color:
 - ‘Color Picker’ dialog box, 7
 - defining, 27–8
 - palettes, 28
 - selecting and editing, 6–8
- Computer display principles, 25–7
- Computer generated imagery (CGI), for animation, 56–64
 - about CGI, 56–7
 - animation process, 62–3
 - arm and leg placing, 58–60
 - ball creation, 58–9
 - Fatcat in Lightwave 7 program, 56, 57
 - hand creation, 58–60
 - importing into Director, 63–4
 - Lightwave modeler, 58–62
 - Non-Uniform Rational B-splines (NURBS) modelers, 57
 - polygon modelers with subdivision surfaces, 57–8
 - ‘Smooth Shift’ technique, 58–9
 - software packages, 223
 - texture maps, 61
 - see also* Modeling low-polygon characters
- Conditions:
 - condition testing, 92–4
 - conditional statements, 93
 - ‘if ... then’ structure, 15–16, 18, 86–7
 - with JavaScript, 88–9
 - with Lingo, 88
 - logical operators, 87–8
 - overlapping rectangles problem, 100–6
 - ‘pointInRect’ example, 95–100
 - truth tables, 93
- ‘copyPixels’ method for scene painting, 214–16
- Cos of an angle, 303–4
- Cursors, custom cursor creation, 154–6
- Custom cursor creation, 154–6
- Custom properties, 17–18
- Database creation:
 - about databases, 182–3
 - adding fields, 183–5
 - adding a table, 183–4
 - basic creation, 183–4
 - inputting data, 186
 - saving the table, 185
- Debugging, 129–39
 - ‘Debugger’ window, 134–9
 - ‘debugPlaybackEnabled’ feature, 139
 - ‘Message’ window, 131–4
 - object inspector, 131
 - problem identification, 129
 - syntax errors, 130–1
 - ‘Trace’ button, 133
- Director:
 - basic interface, 1–2
 - Control Panel window, 2
 - importing animation, 63–4
 - moving to/from paint programs, 32–3
 - simple artwork creation, 1–2
 - see also* Flash MX 2004, exporting to Director
- Distance between points, 89–92
- ‘Edit Favorite Colors’ option, 6
- Embedded fonts, 10–11, 97
- ‘enterFrame’ event, 15
- eps file format, 29, 54
- Euler angles, 244–5, 274–5
- Examples *see* Games; Talking book example; Virtual train set
- ‘Filled Ellipse’ tool, 6
- Flash MX 2004, exporting to Director, 40–5
 - anti-aliasing, 40
 - ‘Button’ option, 141–2
 - dark/white backgrounds, 40–1
 - ‘Dimensions’ and ‘Resolution’, 42
 - Director import options, 42–3
 - exporting as png sequence, 41–2
 - ‘Publish Settings’ dialog, 44
 - saving before publishing, 44
 - symbols, 142
 - see also* Walk cycle (with Flash MX 2004)
- Flash MX 2004, integrating with Director, 140–7
 - about Flash, 140–2
 - array manipulation, 145
 - controlling Director from Flash, 146–7
 - controlling Flash from Director, 145–6
 - project preparation, 142–3
 - sending Lingo commands from Flash, 147
 - use Flash or Director?, 140
 - variable and object handling, 143–5
- Floating point variables, 79–81
- Fonts, 9–11
 - embedded fonts, 10–11, 97

Formats:

- best for backgrounds, 53–4
- best for Director?, 30
- ai file format, 30
- bmp file format, 53
- eps file format, 29, 54
- gif file format, 28–9, 54
- jpeg file format, 29, 53
- png file format, 29, 41–2, 54
- psd file format, 29
- tga file format, 29
- tif file format, 29

Frame loop script, 157–8

Fringing with imported artwork, 156

Functions:

- about functions, 91–2
- creating and using with JavaScript, 76–8, 91–2
- creating and using with Lingo, 76, 91–2

Game, first, 1–19

- ball creation, 2–9
- code adding, 13–19
- improving, 19
- play against the computer, 19
- racket adding, 9–10
- text, 9–13

Games:

- from plan to structure, 117–20
- from structure to project, 120–3
- from project to game, 124–5
- new object creation with JavaScript, 126–8
- new object creation with Lingo, 125–6
- top-down approach, 117
- variables, 117–20
- see also* Board games; Quizzes; 'Tetris' game

Gerald *see* Modeling Gerald (cartoon character)

gif file format, 28–9, 54

Global variables, 72–3

- with virtual train set, 159

Havok Xtra:

- about Havok and rigid body physics, 288–90
- collision tolerance requirements, 291–2
- examples:
 - with keyboard control, 295–8
 - result control, 298–300
 - simple example, 294–5
 - snowboarder game example, 300–1
- initialize method, 291–2
- makeFixedRigidBody functions, 293
- makeMovableRigidBody functions, 292–3
- object methods, 291–3
- object properties, 290–1
- rigidBody functions, 292
- shutdown function, 292
- 'Hither and yon' settings, 249

'if ... then' structure, 15–16, 18, 86–7

see also Conditions

Inker modifier for w3d files, 269

properties, 271

Integer variables, 79–81

JavaScript:

- about JavaScript, 67
 - arrays, 83–5
 - condition statements, 88–9
 - condition testing, 93–4
 - creating and using functions, 76–8
 - distance between points, 90–2
 - Euler angles, 245
 - JavaScript Behavior, 71–2
 - jumping out of loops, 111–15
 - looping (repeating) sections of code, 109–11
 - new object creation, 126–8
 - overlapping rectangles problem, 102–5
 - with platform games, 202–3, 205–7, 208–9, 209–10
 - 'pointInRect' example, 98, 99
 - with quizzes, 186, 187–93
 - rotating models, 245–7
 - scaling objects, 247–8
 - sprite behavior, 74–5
 - strings, 81–3
 - transforming models, 242
 - using JavaScript, 70–1
- jpeg file format, 29, 53

Lasso tool, Flash MX.2004, 35

Layers of graphics:

- Flash MX.2004, 34, 37–8
- see also* Sprites

Level of detail modifiers for w3d files, 266

properties, 270–1

Lights in w3d files, 260, 263

properties, 264–5

Lightwave, 3D modeler, 58–62, 227–30

see also Computer generated imagery (CGI), for animation

Lingo:

- about Lingo, 67
- condition statements, 88
- condition testing, 93–4
- creating a score behavior script, 67–70
- creating and using functions, 76
- custom cursor creation, 154–6
- distance between points, 90–2
- Euler angles, 245
- jumping out of loops, 111–15
- Lingo Movie Script, 71–2
- lists, 83–4
- looping (repeating) sections of code, 109–11
- new object creation, 125–6
- overlapping rectangles problem, 102–4
- 'checkOverlap' function, 106–8

- Lingo: (*Continued*)
 - with platform games, 212–14, 218
 - ‘pointInRect’ example, 97–8, 99–100
 - rotating models, 246–7
 - scaling objects, 248–9
 - sprite behavior, 74
 - strings, 81–3
 - transforming models, 243
- Links via the internet, 315–22
- Lists and arrays:
 - with board games, 166
 - with Flash, 145
 - JavaScript arrays, 83–5
 - Lingo lists, 83–4
- Looping (repeating) sections of code:
 - about looping, 109–11
 - ensuring exit, 113–14
 - frame loop script (for train set), 157–8
 - initialization error problems, 115
 - jumping out of loops, 111–12
 - skipping parts of code, 112–13
 - in ‘Tetris’ game, 123
- Low-polygon characters *see* Computer generated imagery (CGI), for animation; Modeling low-polygon characters
- Luminosity control, 7
- Matrices, 312–13
 - with vectors, 313–14
- Mesh deform modifier for w3d files, 272–3
 - properties, 273
- ‘Message’ window, 131–4
- Model resources in w3d files, 254–7
 - properties, 258–60
- Modeling Charlie (female character):
 - body, 230–2
 - finishing, 233–4
 - hands and feet, 232–3
 - head, 226–30
 - sketches, 224–5
- Modeling Gerald (cartoon character):
 - about Gerald, 235–6
 - bone animating, 238
 - exporting to w3d format, 239
 - full mesh, 235
 - head, 235
 - skeleton, 236–7
 - weight map assigning, 237–9
- Modeling low-polygon characters:
 - Lightwave, 3D modeler, 227–30
 - modeling a head, 223–4, 226–30
 - modeling software, 223
 - principle tools, 224
 - sketches, importance of, 224–5
 - triangles or quads, 225–6
- Models in w3d files, 257
 - properties, 261–2
- Modifiers for w3d files:
 - about modifiers, 266
 - animation modifier, 272
 - collision modifier, 272
 - inker modifiers, 269
 - properties, 271
 - level of detail modifier, 266
 - properties, 270–1
 - mesh deform modifier, 272–3
 - properties, 273
 - subdivision surfaces modifier, 272
 - properties, 272
 - toon modifier, 266
 - properties, 271
- Modularizing, 116, 124–5, 128
 - see also* ‘Tetris’ game
- Motion tweening, 34, 35
- Motions in w3d files, 260
 - properties, 263
- Movie Scripts, 71
- Naming bitmaps, 3–4
- Nesting symbols, Flash MX 2004, 36
- ‘New Behavior’ option, 13–14
- Non-Uniform Rational B-splines (NURBS)
 - modelers, 57
- Objects:
 - with Flash, 143–5
 - object inspector, 131
 - object-oriented programming, 125–8
- Onion-skinning technique, Flash MX 2004, 37–40
- Overlapping rectangles problem, 100–6
 - check overlap function with Lingo, 106–8
- Paint programs, moving to/from Director, 32–3
- ‘Paint’ window, 3, 6
- Palettes, color, 28
- ‘Pencil’ tool, 8
- Physics simulation:
 - rigid body physics, 288–90
 - see also* Havok Xtra
- Pixels, 23–4, 27–8
- Platform games:
 - about platform games, 201
 - ‘beginSprite’ handler, 202–3
 - collision testing, 203
 - ‘copyPixels’ method for scene painting, 214–16
 - ‘enterFrame’ function, 203–7
 - projectile motion, 210–12
 - registration points, 204–5
 - responding to user input, 201
 - scrolling backgrounds, 212–14

- 'setAction' function, 207–9
 - sprites, dynamic creation, 216–18
 - walking along a line of boxes, 209–10
- png file format, 29, 41–2, 54
- Projectile motion, platform games, 210–12
- 'Property Inspector' window, 12
- 'property' key word, 14–15
- psd file format, 29
- 'puppetSprite' method for Director
 - updating, 153–4
- Pythagoras Theorem, 89, 302–3
- Quizzes, 182–200
 - category accessing, 190–2
 - category deletion, 190
 - category listing, 192–3
 - connecting to the database with ASP, 186–9
 - database creation, 182–6
 - front end creation, 193–6
 - implementation ideas, 199–200
 - main loop, 196–9
 - multiple choice or free text?, 182
 - new row creation, 186–9
 - row deletion, 189–90
 - SQL queries/commands, 187–9
- Radians, 306–7
- Resolution considerations, 25–7
- Reversi board game, 164–5
- Right-angled triangles, 89, 302–3
- Rigid body physics, 288–90
 - see also* Havok Xtra
- Root timeline, Flash MX 2004, 36
- Scaling, 27
- Score behavior script, creating with Lingo, 67–70
- Scrolling backgrounds, platform games, 212–14
- Segmenting a character, Flash MX 2004,
 - 34–6, 37–8
- Segments, 1
- Shaders in w3d files, 253–4
 - properties, 255–7
- Sinclair Spectrum computer, 1
- Sine of an angle, 303–4
- Snowboarder game example of Havok Xtra, 300–1
- Sound synchronizing, talking book example, 152–3
- Sprites:
 - dragging sprites, 105–6
 - dynamic creation for platform games, 216–18
 - and layers of graphics, 1
 - sprite behavior, 73–5
 - with JavaScript, 74–5
 - with Lingo, 74
 - sprite locations defining, 16
- SQL queries/commands, quizzes, 187–9
- Strings:
 - with JavaScript, 81–3
 - with Lingo, 81–3
- Subdivision surfaces modifier for w3d files, 272
- properties, 272
- Symbols, Flash MX 2004:
 - with buttons, 141–2
 - positioning, 36
- Synchronizing sound, talking book example, 152–3
- Syntax errors, 130–1
- Talking book example, 151–4
 - clickable events, 152
 - synchronizing sound, 152–3
- Terrains, Aligning to, 284–7
- 'Tetris' game:
 - about 'Tetris', 116–17
 - functionality breakdown, 117
- Text, 9–13
 - color selection, 12
 - font selection, 9–11
 - text cast members, 11–12
- Texture maps, 61
- Textures in w3d files, 252
 - properties, 253
- tga file format, 29
- tif file format, 29
- Tools palette, 4–5
- Toon modifier for w3d files, 266
 - properties, 271
- 'Trace' button, 133
- Tracking with a camera, 280–1
- Train set *see* Virtual train set example
- Transformations, 242–3
 - transformation matrix, 277–80
- Trigonometry:
 - applying trigonometry to Director, 304–8
 - radians, 306–7
 - sin and cos of angles, 303–4
- Variables:
 - about variables, 71
 - absolute value (abs), 81
 - arrays (JavaScript), 83–5
 - decimal values, 79–80
 - floating point values, 79–81
 - global variables, 72–3
 - integer values, 79–81
 - lists (Lingo), 83–4
 - manipulating in Flash and Director, 143–5
 - naming, 79
 - scope of, 71–3
 - strings, 81–3
 - for 'tetris' game, 117–20
 - types of, 78–9

- Vector graphics, 25, 308–14
 - 3D methods:
 - angleBetween, 312
 - cross method, 312
 - distanceTo, 312
 - dot product, 311–12
 - normalize, 311
 - balls moving example, 308–11
 - with matrices, 313–14
 - see also* 3d application
- Virtual train set example, 157–62
 - frame loop script to move train, 157–8
 - global variables used, 159
 - ‘move train’ function, 159–62
- w3d type files for Director:
 - about w3d files, 251–2
 - animation modifier, 272
 - cameras, 263, 265–6
 - properties, 267–9
 - collision modifier, 272
 - inker modifier, 269
 - properties, 271
 - level of detail modifier, 266
 - properties, 270–1
 - lights, 260, 263
 - properties, 264–5
 - mesh deform modifier, 272–3
 - properties, 273
 - model resources, 254–7
 - properties, 258–60
 - models, 257
 - properties, 261–2
 - modifiers, 266–73
 - motions, 260
 - properties, 263
 - shaders, 253–4
 - properties, 255–7
 - subdivision surfaces modifier, 272
 - properties, 272
 - textures, 252
 - properties, 253
 - toon modifier, 266
 - properties, 271
 - Walk cycle (with Flash MX 2004), 34–45
 - adding up and down positions, 39–40
 - bounding box, 35
 - first walk, 36–7
 - lasso tool, 34–5
 - layers, 34, 37–8
 - motion tweening, 34, 35
 - nesting symbols, 36
 - onion-skinning technique, 37–40
 - pivot point/rotation centre, 35–6
 - root timeline, 36
 - segmenting a character, 34–6, 37–8
 - symbol positioning, 36
 - see also* Flash MX 2004, exporting to Director
 - Weight maps, assigning, 237–9
 - ‘z’-buffers, 249, 256–7