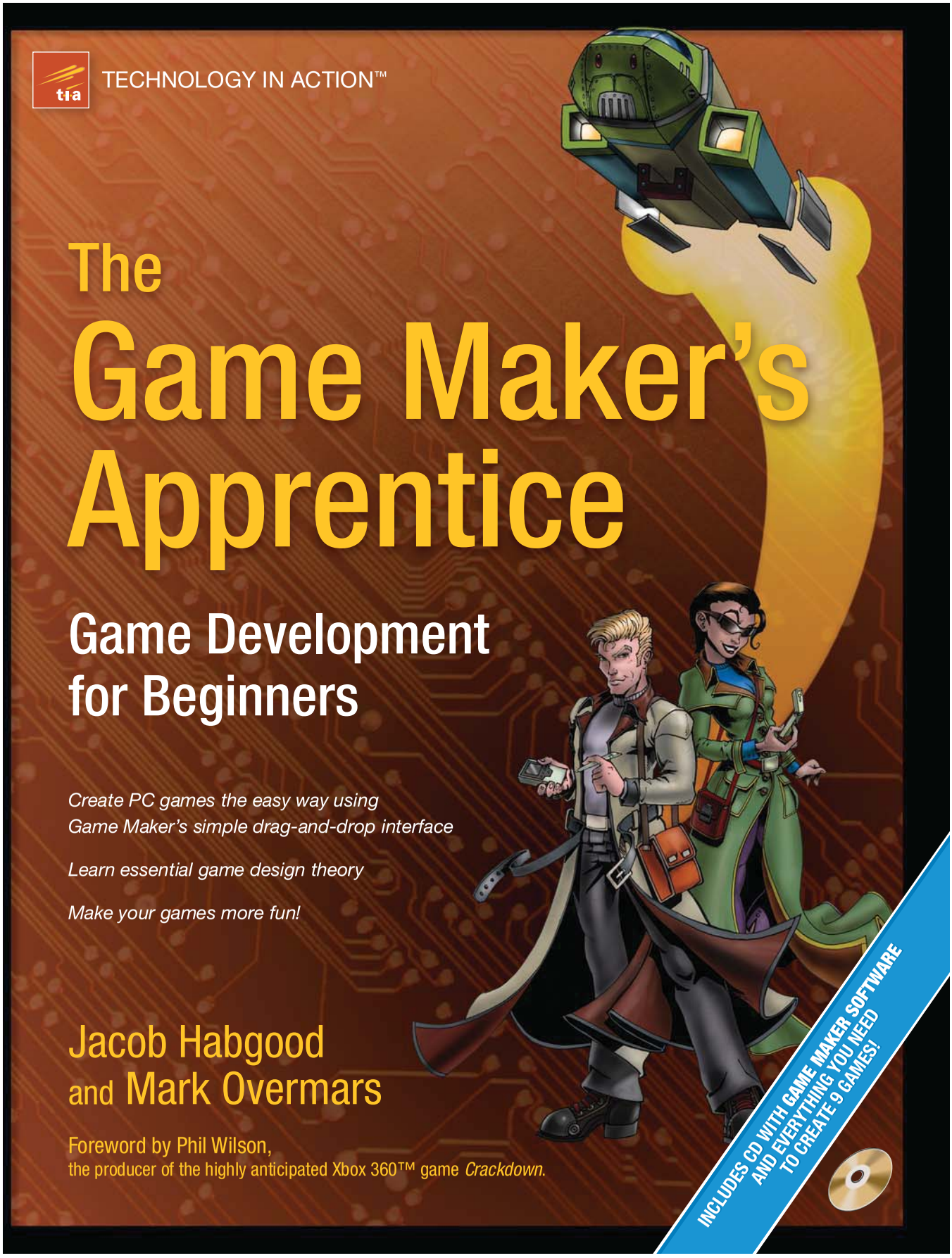
TECHNOLOGY IN ACTION™
The
Game Maker's
Apprentice
Game Development
for Beginners
Create PC games the easy way using
Game Maker's simple drag-and-drop interface
Learn essential game design theory
Make your games more fun!
Jacob Habgood
and Mark Overmars
Foreword by Phil Wilson,
the producer of the highly anticipated Xbox 360™ game Crackdown.
INCLUDES CD WITH GAME MAKER SOFTWARE
AND EVERYTHING YOU NEED
TO CREATE 9 GAMES!

The Game Maker's Apprentice

Game Development for Beginners

Jacob Habgood
Mark Overmars

Apress®

The Game Maker's Apprentice: Game Development for Beginners

ISBN-13 (pbk): 978-1-59059-615-9

ISBN-10 (pbk): 1-59059-615-3

Printed and bound in China 9 8 7 6 5 4 3 2 1

Lead Editor: Chris Mills
Development Editor: Adam Thomas
Technical Reviewer/Additional Material: Sean Davies
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Dina Quan
Proofreader: Lori Bring
Indexer: Present Day Indexing
Artist: Kinetic Publishing Services, LLC
Illustrations and Cover Art: Kevin Crossley
Game Artists: Kevin Crossley, Matty Splatt and Ari Feldman
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

To halcyon days
with a frog,
a parrot,
and a talented bunch of gremlins.

Contents at a Glance

PART 4 Multiplayer Games

PART 5 Enemies and Intelligence

Contents

PART 4 Multiplayer Games

PART 5 Enemies and Intelligence

Foreword

Way back when Mario was still a mere twinkling in Miyamoto's eye, I was the proud owner of a state-of-the-art Commodore 64 microcomputer. It came with a game development system called "The Quill," which allowed anyone to create their own text-based adventure games. It may have been incredibly crude, but it suddenly put at my fingertips the thrill of entertaining my nearest and dearest by devising "interactive challenges" of my own. Unfortunately, I knew little about game design, and rather than easing my players into a new and alien world, I treated them as opponents that had to be defeated before they could reach the end. Their spirits crushed, they left, never to return . . .

It took me years of playing a variety of good (and bad) games to eventually learn how to treat the player to the game-playing experience that their investment of time and money deserved. It took just hours of reading this book to wish I'd had its invaluable guidelines and the accompanying Game Maker tool to help me take my own first steps into game development all those years ago.

Two decades later, I now work for Real Time Worlds as the producer of Crackdown, an imminent Xbox 360 title developed exclusively for Microsoft. Crackdown is the result of over three years of development from a team that's now nearly 70 strong in Dundee (Scotland), with many more contributors across North America and Eastern Europe. This game has cost millions of pounds to create, and already consists of over two and half million lines of programming code! Blood, sweat, and tears have been poured into this title to provide cutting-edge graphics technology, stunning art assets, and dramatic surround sound. We've spent days (and nights) wrestling with new technologies to provide the player with a "playground" and "toy set" that was previously only the stuff of dreams.

Nonetheless, once you strip away the gloss, Crackdown boils down to a handful of gameplay linchpins, or what we term the "pillars of play." Take it from me that when charged with building such a grand gaming monument, it is vitally important to have absolute faith in the basic foundations! I was therefore very pleased to see that this book encourages you to identify these pillars (or game mechanics) and discover how a system of simple rules can combine in unique and compelling ways to create a spellbinding experience.

As you progress through the book you'll build a series of excellent games that you might never have even dreamed you could be capable of creating right now. The instructions are clear and concise, but also encourage you to experiment with your own designs. For example, your version of the captivating and original Koalabr8 game (Chapter 7) will almost certainly be a unique piece of software. The crazy devices you invent, and the way you lay out your levels, will certainly differ from mine. Watch out for Lazarus too (Chapter 4)—it may interest you to know that this eponymous hero first appeared in Jacob's student portfolio, and was partly responsible for securing his first programming job in the industry!

Mark and Jacob have brought together decades of game development expertise in this book. As well as being a professor of computing, and the creator of Game Maker, Mark first cut his game-programming teeth creating versions of games like Super Breakout for the Atari ST. Jacob has a string of titles to his name, and his in-depth knowledge of "the craft" consistently yields outstanding results. Never more so was this the case than when I had the pleasure of working with him on the team that created the PlayStation hit Hogs of War (also mentioned in this book). Where Jacob differs from his peers is in his mastery of all four of the fundamental game development disciplines: programming, sound, art, and, of course, design. Now, thanks to Mark's Game Maker software, you can find out what it feels like in their world!

One of the key messages I hope you'll take away from this book is that there's a world of difference between having a great idea for a game and being a great game designer. The initial idea is simply the seed from which the game grows, or the stone from which the pillars are hewn. The role of a designer is to fully realize the vision: conceiving and continually refining the various supporting mechanisms to make them mesh like the components of a Swiss timepiece. As is repeatedly stated in these pages, there is no correct solution to game design—only a great idea, well executed and injected with personal flair and enthusiasm. Even if you're struggling to pin down that idea right now, I'm sure you will have wrestled it onto the screen and into the hands of friends and family before finishing the final chapter of *The Game Maker's Apprentice*.

Good luck!

Phil Wilson
Producer, Real Time Worlds

Crackdown

About the Authors

JACOB HABGOOD is 30 years old and has been writing computer games since he was 10. He wanted to be a psychologist when he grew up, but somehow he ended up with a computer science degree and went into the games industry instead. He worked as a professional game developer for seven years, programming console games for Gremlin Interactive and Infogrames/Atari in the north of England. During this time he contributed to a range of successful titles and led the programming teams on Micro Machines (PlayStation 2, Xbox, and Nintendo GameCube) and Hogs of War (PlayStation).

Jacob is now a doctoral student at the University of Nottingham, researching the educational potential of computer games. As part of this research, Jacob runs clubs and workshops teaching children and teenagers how to make their own computer games and provides free teaching resources through his website: `gamelearning.net`. All being well, this work will soon earn him a Ph.D. from Nottingham's Department of Psychology so that he can finally consider himself grown up.

MARK OVERMARS is a full professor in computer science at Utrecht University in the Netherlands. There he heads the research center for Advanced Gaming and Simulation (`www.gameresearch.nl`) in which researchers from different disciplines collaborate on all aspects of gaming and simulation. One of Mark's prime research domains is computer games. He is also one of the founders of the Utrecht Platform for Game Education and Research (`www.upgear.nl`), a collaboration of different game-related educational programs in the Netherlands. For many years he has taught courses on computer game design at Utrecht University, and has given lectures on game design to many types of people (high school kids, teachers, researchers, and politicians). Mark is the author of a number of popular software packages, in particular, the Game Maker software package used as the development tool in this book.

About the Technical Reviewer

SEAN DAVIES is 28 years old and has been fascinated by computer games from an early age. He grew up fairly certain that he would become a novelist—or possibly a rock star, but eventually came to a number of important realizations:

1. He's probably never going to be a rock star.
2. Game programming is quite cool, though.
3. Badgers are just really big weasels (the exchange rate is approximately 20 weasels to the badger if you're interested).
4. Any attempts to construct a serious calculus of the family Mustelidae are probably best kept to yourself—people think you're strange (see above).

Having made these startling realizations at such an early stage, the rest of his career path was pretty much decided. After graduating with a degree in computer science, he joined Infogrames in Sheffield, UK, and has worked in the games industry ever since. When Infogrames Sheffield closed its doors in 2002, he joined Sumo Digital, where he still is today. Sean is currently Xbox platform lead on Outrun 2006: Coast to Coast, which Sumo is developing for SEGA, and is looking forward to working on some next-generation console programming in the near future. He currently has no intention at all of ever becoming properly grown up.

Outrun 2006: Coast to Coast

About the Illustrator

The first things **Kev Crossley** remembers drawing as a child were some Daleks and the Incredible Hulk, and he knew from that point on that he would grow up to be one or the other. When Kev was five, his dad brought home the videogame Pong, and Kev has been trying to come to grips with it ever since. Nonetheless, he has managed to get through a couple of Zelda games and has spent many a happy hour blasting the head off one of this book's authors with a Rail Gun.

Kev spent some time in a university, but eventually realized that an art degree was not going to give him access to Time Lords or gamma rays, so he decided to work in a bakery instead. Eventually he got sick of eating coconut macaroons and biscuits and applied for a job making videogames for a little green monster. For the next third of his life, he had a great time producing graphics and animation for over 20 titles—some of which were quite good.

These days Kev is a concept artist at Core Design in Derby, UK, where he has occasionally been known to hang out with women obsessed with tombs. He has also done copious amounts of freelance illustration and writing for publishers all over the world, including a series of instructional drawing books and sequential work for Rebellion's sci-fi comic *2000AD*. His one regret is that he can't ride a skateboard, because a cross between the Daleks and Tony Hawk would be unstoppable.

Acknowledgments

By rights, this book really shouldn't exist, because it's required far too much effort from far too many people to make it a profitable endeavor. Nonetheless, it does exist, and as a result there are a lot of people who need to receive our heartfelt thanks in helping us to realize this labor of love.

First and foremost, we need to thank all those people closest to the individuals directly involved in bringing this book into existence. Their influence may not be obvious to the reader, but projects like this could never happen without the support and understanding of the wives, girlfriends, and families of all the people involved in this book. In particular, Jacob would like to thank Jenny, Fiona, Michelle, and Amelia, all of whose names should adorn the credits of many a videogame for the sacrifices that they regularly make to indulge the creative passions of their loved ones.

The next biggest thanks should go to Matty Splatt, who has shied away from a full billing in the "About the" sections but did a fantastic job of bringing Koalabr8 and Pyramid Panic to life with his comical and beautifully polished graphics. We would also like to give a special thanks to Ari Feldman for allowing us to use and modify his game sprites for the Wingman Sam game.

More thanks go to Jenny and Marguerite Habgood for their grammatical critiques, and the following people for all their comments and input into the project: Sarah Peacock, Judy Robertson, John Sear, and Phil Wilson. Additional thanks goes to all the staff and students at Sheffield West City Learning Centre, who suffered the book's instruction in its earliest form.

Quick thanks also to everyone who enthusiastically play-tested the games in the book, including Gail Clipson, Fiona Crossley, Katie Fraser, Giulia Gelmini, Jasmin Habgood, Martijn Overmars, Ronald Overmars, and Stuart Reeves.

Finally, we would like to thank everyone at Apress for their support, and for sticking with us even after it became plainly obvious that both authors were far too busy to write a book!

Introduction

Who wouldn't want to make computer games? It's creative, rewarding, and these days even pretty darn cool too. You can make them to share with your school friends, your work colleagues, your grandchildren, or even the entire gaming world. This book is not specifically for the young or old, but anyone who loves computer games and wants to have a go at making them for themselves. We've all painted a picture, written a story, and made a wobbly piece of pottery at some point in our lives, so it's now time to embrace the art form of the future and try making computer games too.

This book provides a collection of engaging tutorials that introduce you to the Game Maker tool and teach you how to use it. The first four parts of the book take you step by step through seven different projects using Game Maker's simple drag-and-drop programming system. By the time you've finished making Evil Clutches, Galactic Mail, Lazarus, Super Rainbow Reef, Koalabr8, Wingman Sam, and Tank War, you'll have a well-rounded experience of making games with Game Maker. Parts 2, 3, and 4 also end with game design chapters that encourage you to stand back from your creations and consider how principles of game design can be used to make them more fun. Moreover, we don't just talk about it, but we provide new versions of the games with improved features so that you can experience for yourself how solid game design can lead to good gameplay.

Game Maker provides a simple environment that allows complete beginners to quickly start building games, using an icon-based system of events and actions (see Figure 1). This drag-and-drop programming technique provides an easy way to learn about game development and allows you to create complete games without going near a traditional programming language.

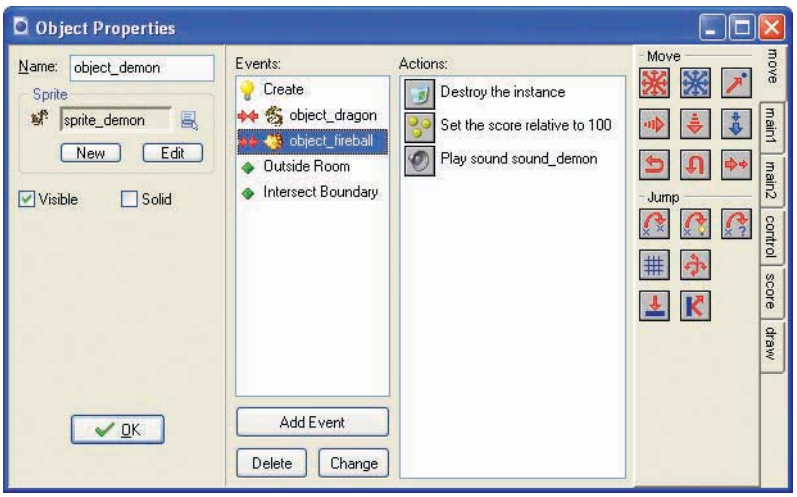

Figure 1. *Game Maker's simple drag-and-drop system uses iconic events and actions to program computer games.*

However, once you become more experienced, traditional languages can provide a more powerful way to program games. Consequently, Game Maker also provides the Game Maker Language (GML), which underpins Game Maker and makes it such a powerful tool (see Figure 2). The last part of the book uses several simple examples to introduce you to GML, before we demonstrate how you can use it to create artificial intelligence for undead creatures in the Pyramid Panic game.

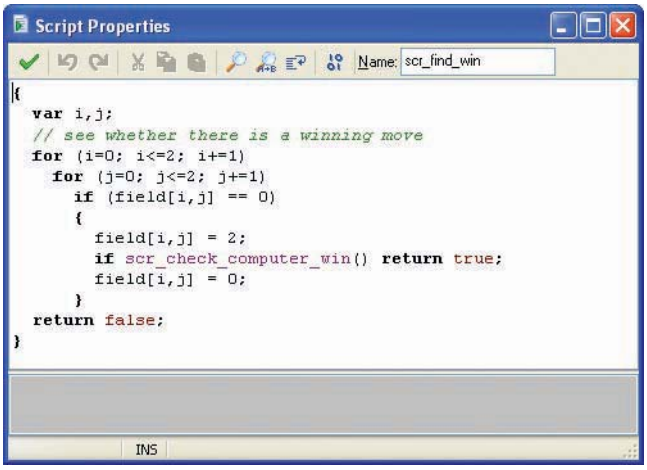

Figure 2. *Game Maker Language (GML) provides extra power for advanced users.*

The example games in this book have been brought to life with graphics and illustrations by real games industry artists. Furthermore, you can use all the professional resources provided on the CD in your own Game Maker projects with the blessing of the publisher and authors. We only ask that you share your creations with the online Game Maker community so that we can see what you have created with them. We want you to enjoy the creative journey ahead and hope that it will help you to share in our passion and enthusiasm for creating computer games!

PART 1

Getting Started

Welcome to the world of game development. Playing games can be a lot of fun, but you're about to discover why making them is so much better!

CHAPTER 1

Welcome to Game Maker

If you're looking for an enjoyable way to learn how to make computer games, then this is the book for you. You don't need a degree in computer science and you won't have to read a book the size of a telephone directory—everything you need is right here. As long as you can use Windows without breaking into a cold sweat, you have all the qualifications you need to start making your own games. In the chapters ahead, we'll show you how to make nine complete games and pass on some of our hard-earned professional experience in game design along the way. Already, you are just two chapters away from completing your first game and have taken your first step along the path of the game maker's apprentice!

Every trade has its tools, and every tradesman knows how to choose the right tool for the job. In this book we will be creating all the games using a software tool for Windows called *Game Maker*. Game Maker is ideal for learning game development as it allows you to start making games without having to study a completely new language. This makes the whole learning experience a lot easier and allows you to concentrate on creating great game designs rather than getting bogged down with the technicalities of programming. Nonetheless, programming languages can offer many advantages to experienced users and so Game Maker also includes its own language, which is there for you to discover when you feel ready to use it.

You'll be pleased to hear that there is a free version of Game Maker included on the CD accompanying this book. All the games can be made using this free version; however, there are some special effects in the later chapters that will only work with a registered version of Game Maker. If you want to register your version for this, or any of the other extra features that it unlocks, then it can be obtained from the Game Maker website for a very small fee (currently US $20): `www.gamemaker.nl`.

Installing the Software

You can't begin making the games in this book until you have the Game Maker software installed on your PC. You'll find the install program in the `Program` folder on the CD, so insert the disc, navigate to the `Program` folder, and start the program called `gmaker_inst.exe`. The form shown in Figure 1-1 should then appear on the screen.

Figure 1-1. *Install Game Maker by following the instructions on the screen.*

Click **Next** and follow the instructions as they appear. We strongly recommend installing the program in the default directory.

■ **Note** Game Maker requires a fairly modern PC running Windows 98SE, Me, 2000, or XP—although Windows XP is preferable. You'll need a DirectX-compatible graphics card with at least 32MB of memory and DirectX 8 or later installed on your machine. A DirectX-compatible sound card is also required for sound and music. If your machine does not meet these requirements, you might have problems running Game Maker or the games created with it. Don't worry if all this techno-babble makes no sense—just try it out because you're not likely to have any problems unless your PC is very old. See the `readme` file in the `Program` folder on the CD for further details.

Game Maker should start automatically once the installation has completed. You can also launch it directly from the Windows Start menu or by double-clicking the Game Maker icon on your desktop. The first time you run Game Maker on a new computer, you will be asked if you want to run the program in Advanced mode (see Figure 1-2). Click **No** as it will be easier to stick with the Simple mode for the time being—we'll show you how to switch to Advanced mode later on.

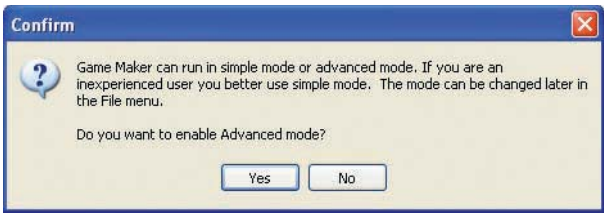

Figure 1-2. *Click* ***No*** *at the prompt, as we want to start by using Simple mode.*

Registration

The free version of Game Maker provided with this book is fine for learning how to make all the games in this book, but some of Game Maker's more exciting features are disabled unless you register the program. Registering will also allow you to create more professional-looking games by removing the Game Maker pop-up message that appears at the start of any game. Until you register, a reminder message will appear from time to time like the one shown in Figure 1-3. Clicking **Close this Form** will make it disappear, but if you use Game Maker a lot we strongly encourage you to register it. Registration will support the further development of Game Maker and ensure that everyone who uses it can continue to enjoy making games for years to come. Information about registration can be obtained by clicking **Go to Registration Webpage** or by simply visiting the Game Maker website at `www.gamemaker.nl`.

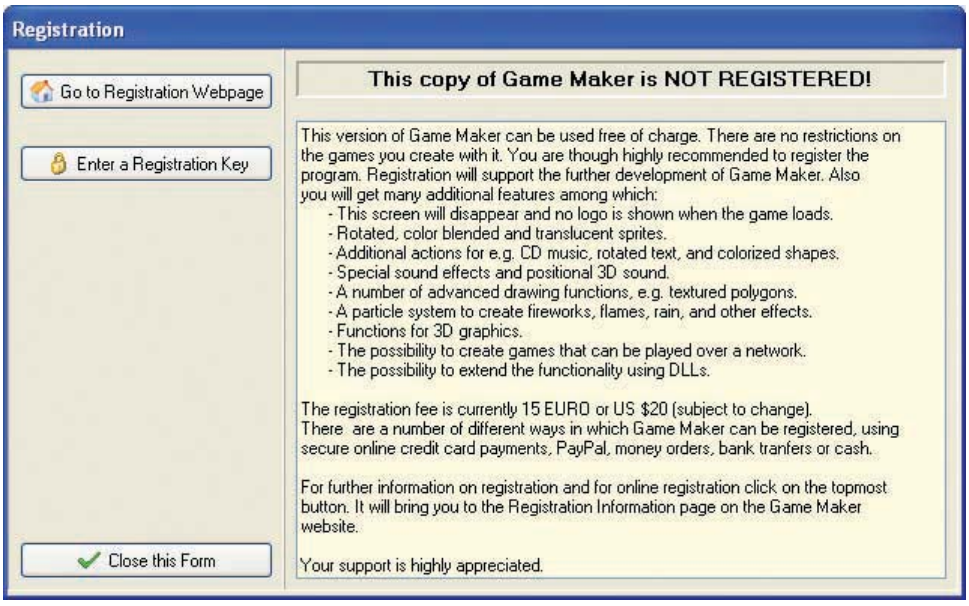

Figure 1-3. *Select* ***Close this Form*** *to start working with Game Maker or* ***Go to Registration Webpage*** *to learn more about registration.*

The Global User Interface

If everything has gone according to plan, then you should now be looking at the window shown in Figure 1-4. If not, consult the `readme` file in the `Program` folder on the CD for possible causes and further instructions.

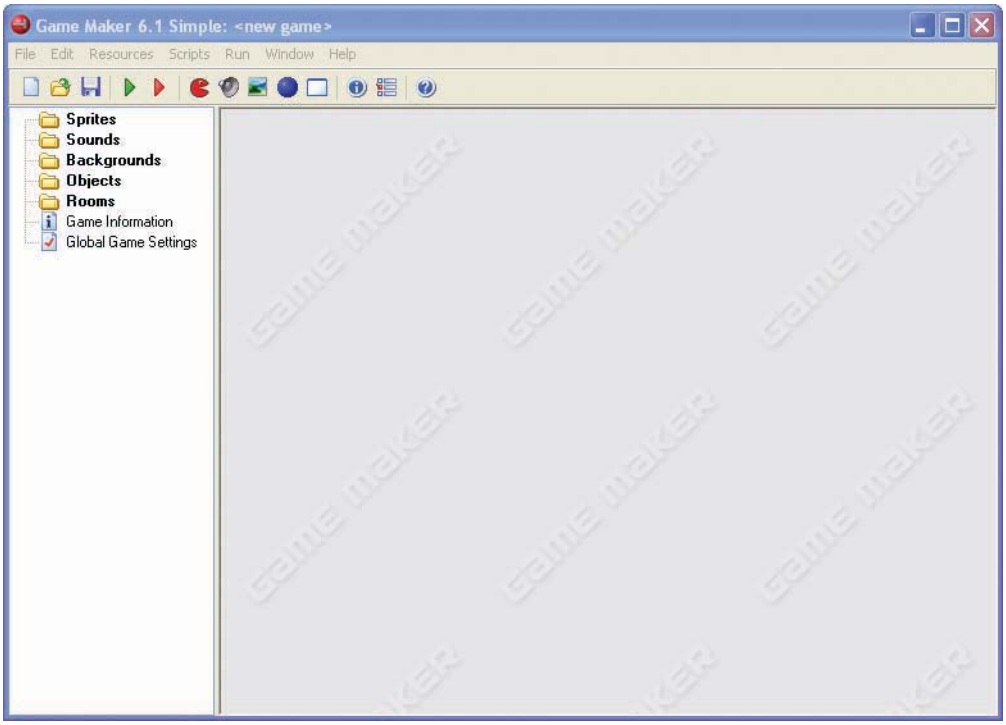

Figure 1-4. *This is Game Maker's main window.*

We'll describe the user interface in more detail in the next chapter, but for the moment just notice that there's a standard-looking menu and toolbar at the top of the screen, and a *folder tree* on the left-hand side. This tree is where we will add all the different *game resources* that are used to make Game Maker games. More about resources later, but first let's make sure Game Maker is working properly by running a simple game.

Running a Game

Loading and running a game that has been created using Game Maker couldn't be simpler. Just complete the following instructions.

Running the sample game:

1. Click the **File** menu and select **Open** from the drop-down menu. This will bring up the standard Windows file requester.
2. Make sure the CD is in your CD drive. Navigate to the `Games/Chapter01` folder on the CD and look for `bouncing.gm6` (all Game Maker files end with this `.gm6` extension). Select this file and click **Open**.

It may not seem as if anything has changed, but if you look carefully, there are now plus signs in front of the different folders on the left-hand side. Clicking these plus signs will open up the folders to show the resources that they contain.

Let's run the game. Don't expect too much, though; this is just a simple demo to check whether Game Maker works correctly on your machine. Click the green play button on the toolbar. The Game Maker window should disappear and the image in Figure 1-5 will appear. This is the pop-up that is only shown in the free version of the program and can be removed by registering your copy of Game Maker.

Figure 1-5. *This pop-up message appears in the free version of Game Maker.*

After a short pause, a game window should appear like the one in Figure 1-6. You should hear music and see a number of balls bouncing around the screen. If you like, you can try to destroy the balls by clicking on them with the left mouse button, or you can press F4 to make the game window fill the screen. When you have seen enough, press the Esc key to end the game.

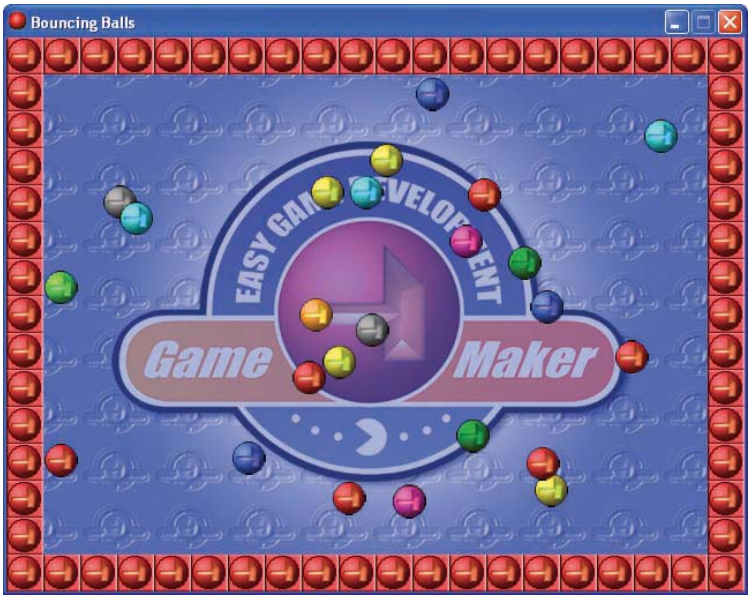

Figure 1-6. *In the bouncing ball demo, you can try to destroy the balls by clicking on them with the left mouse button.*

If something went wrong (for example, you got an error message or you didn't hear the music), then consult the `readme` document in the `Program` folder on the CD for possible causes and further instructions. You can now close Game Maker by choosing **Exit** from the **File** menu.

How to Get More Information

This book will show you how to make some cool games with Game Maker, but it is not a complete manual for everything that Game Maker can do. Fortunately, the Game Maker help file contains all the facts, and you can access it at any time through the program's **Help** menu or the Windows Start menu. A copy of this document is also available in the `Documents` folder on the CD.

You are also strongly advised to check out the official Game Maker website at `www.gamemaker.nl`. Here you'll find the latest version of the program and lots of games that have been created with it—as well as additional resources and documents. The website also gives you access to the Game Maker user forum. This is a very active forum and a great place to get help from other users or just to exchange ideas and games.

What's Next?

Now that the boring stuff is out of the way, so I think it's about time we made our first game. You're probably thinking your first game will be pretty dull—something with more bouncing balls, perhaps? Not likely!

CHAPTER 2

Your First Game: Devilishly Easy

Learning something new is always a little daunting at first, but things will start to become familiar in no time. In fact, by the end of this chapter, you'll have completed your very first gaming masterpiece!

Designing the Game: Evil Clutches

Before you start making a game, it's a good idea to have an idea of what you're aiming for. Commercial game developers usually prepare long design documents before they start creating a game. Nonetheless, writing documents isn't a fun way to learn how to make games, so we'll keep our designs as short as possible. We're calling the game in this chapter *Evil Clutches*, and this is its design:

You play a mother dragon who must rescue her hatchlings from an unpleasant band of demons that have kidnapped them. The band's boss sends a stream of demons to destroy the dragon as the hatchlings make their escape. The mother can fend off the boss's minions by shooting fireballs, but must be careful not to accidentally shoot the hatchlings!

The arrow keys will move the dragon up and down and the spacebar will shoot fireballs. The player will gain points for shooting demons and rescuing young dragons, but will lose points for any hatchlings that accidentally get shot. The game is over if the dragon is hit by a demon, and a high-score table will be displayed. Figure 2-1 shows an impression of what the final game will look like.

Using this description, we can list all the different elements needed to create our game: a dragon, a boss, demons, hatchlings, and fireballs. Making the game will require pictures of each of these as well as a background image, some sound effects, and music. We call all these different parts that make up the game *resources*, and the resources for this game have already been created for you in the `Resources/Chapter02` folder on the CD. For the remainder of the chapter, we will learn how to put these resources together into a game and bring them to life.

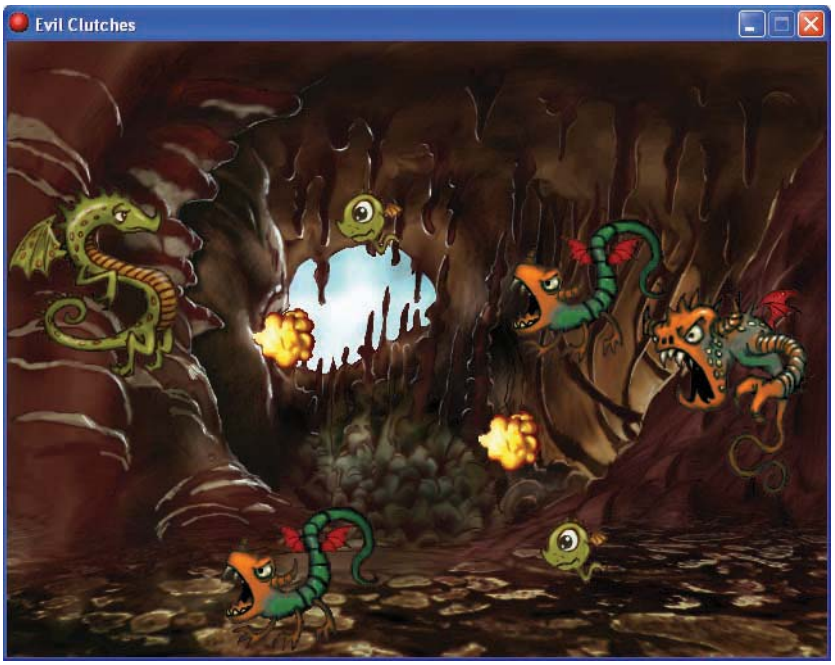

Figure 2-1. *Here's the Evil Clutches game in action.*

Sprites

In Game Maker, pictures of dragons, demons, and other game objects are all called *sprites*. Sprites are one kind of resource used in games, and they can be made from images that have been created in art packages or downloaded from the Internet. Game Maker includes a simple sprite editor for drawing your own sprites, but you can use any drawing package you like for this purpose. However, creating sprites is time consuming, so we've already provided professionally drawn sprites for each game.

If you've not done so already, start up Game Maker. Figure 2-2 shows the (rather empty) main window that appears.

■ **Note** If your window doesn't look exactly the same as shown in Figure 2-2, then you're probably running Game Maker in Advanced mode. To switch to Simple mode, choose **Advanced Mode** from the **File** menu and the checkmark beside it will disappear.

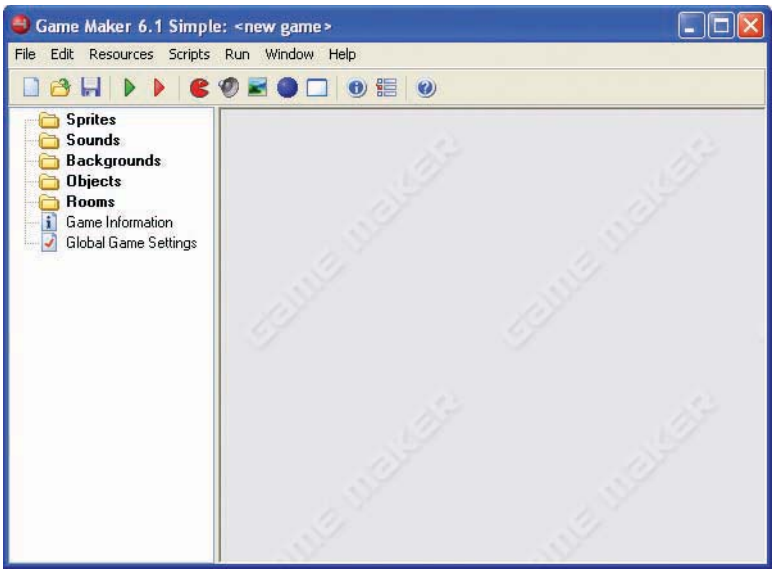

Figure 2-2. *In the main window of Game Maker (in Simple mode), the menu and toolbar runs along the top of the window and a list of resources down the left side.*

The left side of the window shows the different types of resources that make up the game: sprites, backgrounds, sounds, and so forth. These are currently empty, but the names of new resources will appear here as they are added to the game. The menu bar along the top of the window contains all the commands that allow us to work with resources—although most common tasks can also be accessed using the buttons on the toolbar. We'll begin by using the **Create Sprite** command to create a new sprite.

Creating a new sprite resource for the game:

1. From the **Resources** menu, choose **Create Sprite**. The Sprite Properties form appears, like the one shown in Figure 2-3.

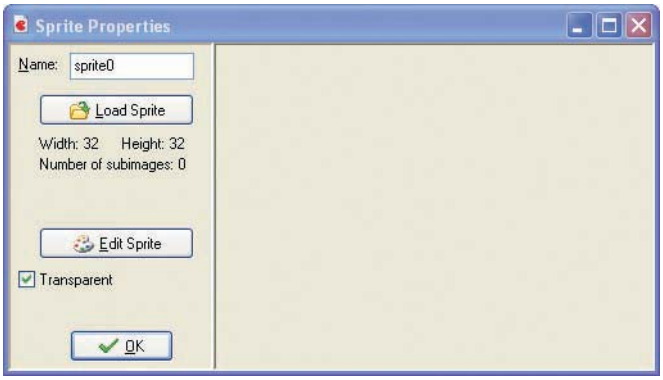

Figure 2-3. *Open the Sprite Properties form for a new sprite.*

2. Click the **Name** field, where it currently says `sprite0`. This is the default name created by Game Maker for the new sprite, but you should rename it to `sprite_dragon`.
3. Click the **Load Sprite** button. This opens the standard Windows file requester.
4. Select the required image using the file requester. The image for the dragon is called `Dragon.gif`, and you'll find it in the `Resources/Chapter02` folder on the CD. Your Sprite Properties form should now look like Figure 2-4.

■ **Note** Always avoid using spaces and punctuation in names for resources as they will confuse Game Maker when you try to use some of its more advanced functions later on. You can use the underscore (_) symbol instead of spaces, which is usually found on the same key as the minus symbol (press Shift and the minus key).

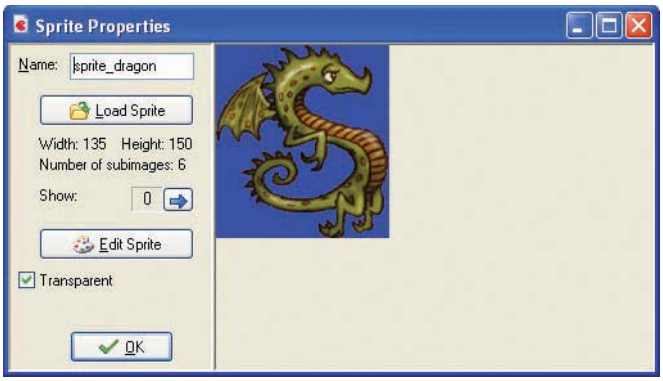

Figure 2-4. *The Sprite Properties form looks like this after we load the dragon sprite.*

5. Click **OK** to close the form. You have now created a sprite.

The dragon sprite should now have been added to the list of sprites in the resource list. If you ever need to change a resource, you can reopen its properties form by double-clicking on its name in the resource list. Do this now and take another look at the dragon sprite's properties (Figure 2-4).

The form shows that there are six subimages to this sprite. Sprites often consist of several images shown one after the other to create the illusion of movement. If you move through the subimages using the blue arrow button, you will notice that there are actually only two different images for this sprite. The extra copies make sure that the dragon doesn't flap its wings too quickly when it's animating.

The checkmark next to the **Transparent** property means that the background of the dragon sprite is see-through. Most sprites are set to transparent so that the surrounding rectangle won't be drawn when the sprite appears in the game. Figure 2-5 shows the difference that the **Transparent** option makes—the advantages are obvious to see!

Figure 2-5. *Here's the dragon sprite with the Transparent option set (left) compared to the same dragon without the Transparent option (right).*

■ **Note** Game Maker works out which color to make transparent based on the color in the bottom leftmost corner of each image. This is worth remembering when you want to create your own sprites.

Okay, let's create the other sprites for the game in the same way.

Creating the remaining Evil Clutches sprites:

1. From the **Resources** menu, choose **Create Sprite**.
2. In the **Name** field in the Sprite Properties form, type the name `sprite_boss`.
3. Click the **Load Sprite** button and choose the file `Boss.gif`.
4. Click **OK** to close the Sprite Properties form.
5. Now create a demon sprite, baby sprite, and fireball sprite using the `Demon.gif`, `Baby.gif`, and `Fireball.gif` files in the same way. Give each sprite an appropriate name (using only letters and the underscore symbol).

This completes all the sprites needed to create the Evil Clutches game.

Objects

Sprites don't do anything on their own; they just store pictures of the different elements in the game. *Objects* are the parts of the game that control how these elements move around and react to each other. We'll begin by creating our first object to tell Game Maker how we want the demon boss to behave.

The Boss Object

The following steps create a new object and assign it a sprite so that Game Maker knows how it should look on the screen.

Creating a new object and assigning it a sprite:

1. From the **Resources** menu, choose **Create Object**. An Object Properties form like the one in Figure 2-6 appears.

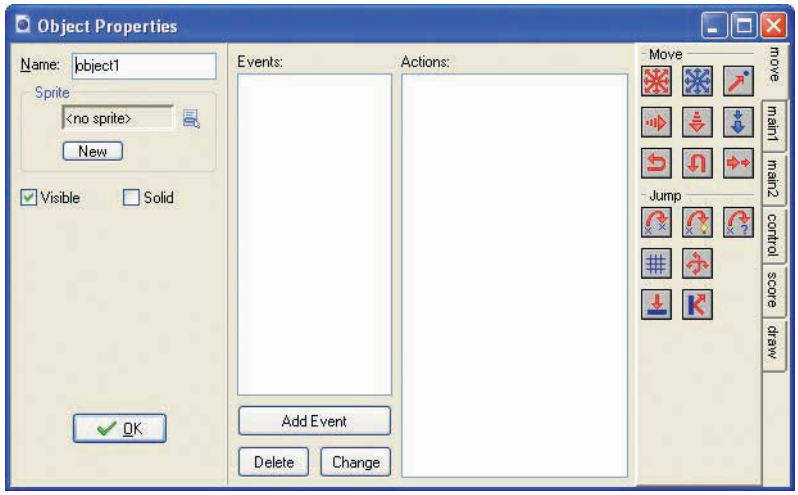

Figure 2-6. *Open the Object Properties form for your new object.*

2. In the **Name** field, give the object a name. You should call this one `object_boss`.
3. Click the icon at the end of the sprite field and a list of all the available sprites will appear. Select the `sprite_boss` sprite.

■ **Caution** Always make sure that you give your object resources names that are different from your sprite resources. Ending up with an object and a sprite both called "dragon," or two objects called "demon," can confuse Game Maker when you try to use its more advanced functions later on. Adding prefixes like "sprite_" or "object_" to names is a good way to achieve this without having to think of new names.

Events and Actions

Game Maker uses *events* and *actions* to specify how objects should behave. *Events* are important things that happen in the game, such as when objects collide or when the player presses a key on the keyboard. *Actions* are things that happen in response to an event, such as changing an object's direction, setting the score, or playing a sound. Game Maker games are basically just a collection of objects with actions to tell them how they should react to different events. Therefore, to set the behavior of an object in Game Maker you must define which events the object should react to and what actions they should perform in response.

The boss object's lists of events and actions are currently empty. We're going to begin by adding an event and action that will start the boss moving up the screen at the beginning of

the game. This will be complemented by an action that reverses the vertical direction of the boss in the event that it collides with the edge of the screen. As a result, the boss will continually move up and down between the top and bottom of the screen.

Adding a create event for the boss object:

1. Click the **Add Event** button. The Event Selector appears, as shown in Figure 2-7.

Figure 2-7. *Click **Add Event** to open the Event Selector pop-up form.*

2. Click the **Create** event to add it to the list of events. A new event is automatically selected (with a blue highlight) in the event list, as shown in Figure 2-8. This means we're already looking at this event's **Actions** list alongside it (which is currently empty).

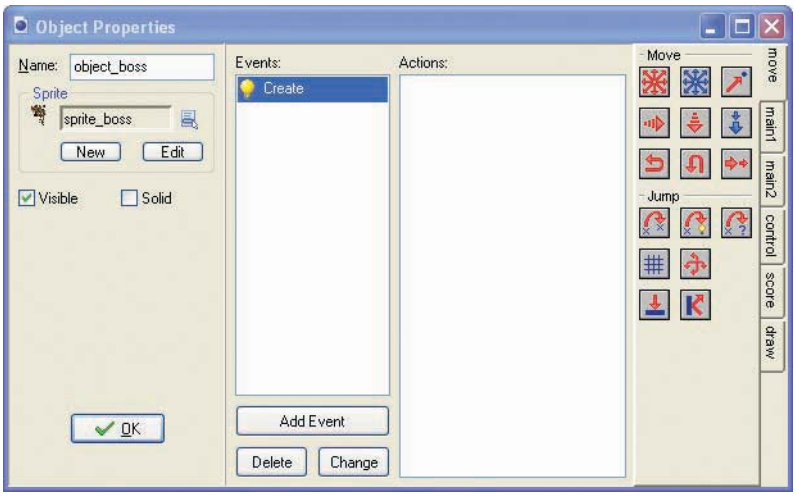

Figure 2-8. *This is how the Object Properties form should look once the name, sprite, and **Create** event have been added.*

3. Next you need to include the **Move Fixed** action in the list of actions. To do this, press and hold the left mouse button on the action image with eight red arrows, drag it to the empty **Actions** list box, and release the mouse button. An action form will then pop up asking for particular information about this action (see Figure 2-9).

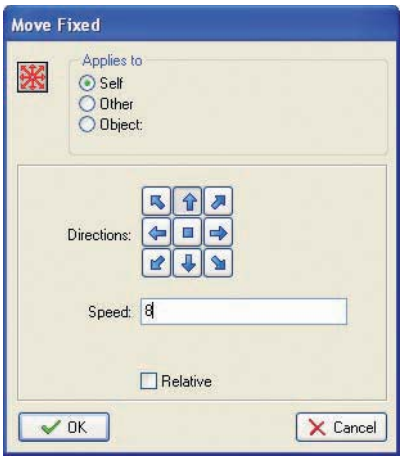

Figure 2-9. *Here's the action form for the* ***Move Fixed*** *action.*

■ **Note** Whenever we use an action in the instructions, that action's image is shown in the left margin to help you find the correct one.

---

4. Select the up arrow and enter a value of 8 for the **Speed**. This will make the object move vertically 8 pixels (the tiny squares that make up a monitor display) for every step that it takes.
5. Press **OK** to close the action form and it will be included in the list of actions.

■ **Note** All of Game Maker's actions are organized into tabbed pages of icons on the right of the **Actions** list. Browse through the different action tabs to see all the various actions and hold your mouse over one to reveal its name.

This event should start the boss moving upward. Now we'll add an event to reverse an object's vertical direction when it collides with the edge of the screen. This event is called the Intersect Boundary event because it gets called when the object's sprite intersects the screen's boundary by being partly in and partly out of the screen.

Adding an intersect boundary event for the boss object:

1. Click the **Add Event** button.
2. Choose **Other** from the Event Selector pop-up form and select **Intersect boundary** from the drop-down menu that appears. This action will then be added and selected in the list of events.

3. Include the **Reverse Vertical** action in the list of actions for this event. You'll now see the form shown in Figure 2-10.

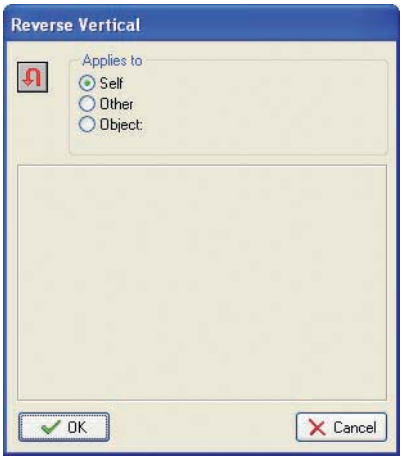

Figure 2-10. *The action form for the* ***Reverse Direction*** *action looks like this.*

4. Nothing needs changing on this form, so just click **OK**. The Object Properties form for the boss object now looks like the one shown in Figure 2-11.

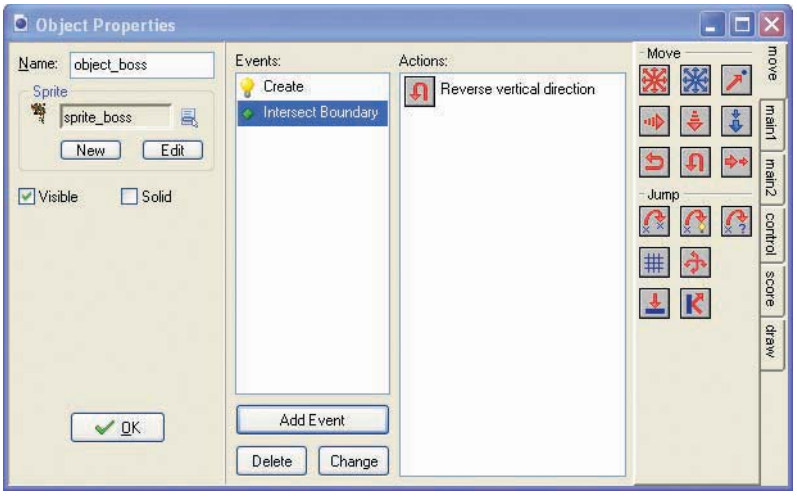

Figure 2-11. *In the Object Properties form for the boss object, we've added two events, along with their corresponding actions.*

These are all the events and actions we need for the boss right now. You can switch between the different events by clicking on them in the **Events** list. The selected event is highlighted in blue and the actions for that event are then shown in the **Actions** list. You can edit the properties of each action by double-clicking on them, but we're done with the boss object for now.

5. Click **OK** at the bottom left of the form to close it.

The Dragon Object

Now let's turn our attention to the heroine of the game. We'll begin by creating an object for the dragon in the same way as for the boss.

Creating a dragon object:

1. From the **Resources** menu, choose **Create Object**.
2. Give the object a name by entering `object_dragon` in the **Name** field.
3. Select the `sprite_dragon` sprite from the drop-down sprite menu.

The dragon also needs actions to make it move up and down the screen, but this time only when the appropriate keys are pressed on the keyboard. We do this by using keyboard events.

Adding keyboard events for the dragon object:

1. Click the **Add Event** button.
2. Choose a **Keyboard** event and select **<Up>** from the pop-up menu (to indicate the up arrow key).

3. Include the **Move Fixed** action in the **Actions** list.
4. In the action form, select the upward direction and set **Speed** to `16`.

5. Repeat the previous process to add a **Keyboard** event for the **<Down>** key that includes a **Move Fixed** action with a downward direction and a speed of `16`. The Object Properties form should now look like the one shown in Figure 2-12.

We just need one more event and action to make the dragon's movement work correctly. Our **Keyboard** events will start the dragon moving when the player presses the arrow keys, but there are currently no events to stop it from moving again when the keys are no longer being pressed. We use the **Keyboard, <no key>** event to test for when the player is no longer pressing any keys.

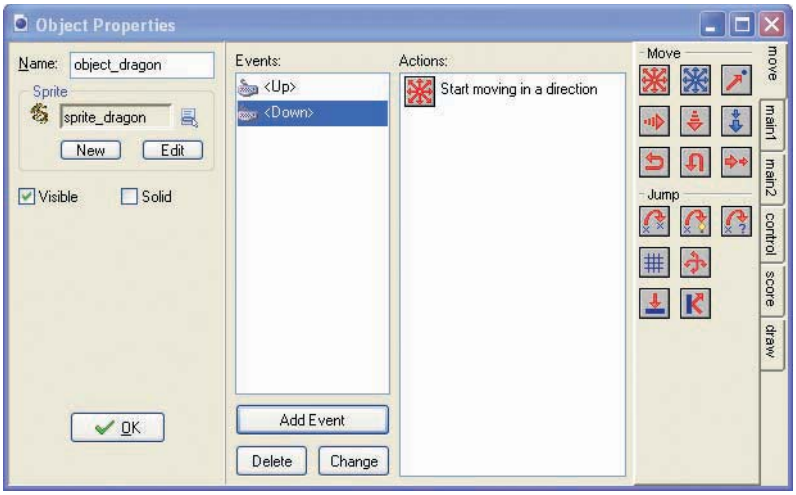

Figure 2-12. *The Object Properties form for the dragon looks like this once we add the* ***<Up>*** *and* ***<Down>*** *events.*

Adding a no key event for the dragon object:

1. Click the **Add Event** button.
2. Choose a **Keyboard** event and select **<no key>** from the pop-up menu.
3. Include the **Move Fixed** action in the **Actions** list for this event.
4. Select the center square in the action form, to indicate no movement, and set **Speed** to 0. The form should now look like Figure 2-13.

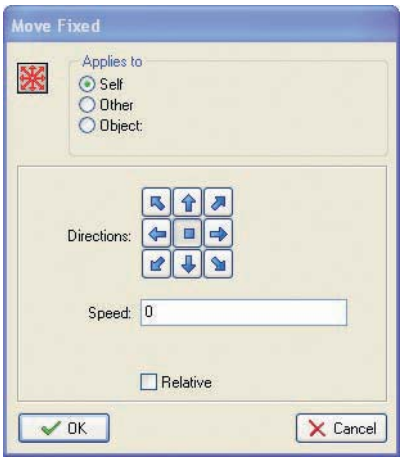

Figure 2-13. *These settings stop the movement of the dragon.*

5. That's all the actions we need to make our dragon move up and down, so click **OK** to close the Object Properties form for the dragon object.

■ **Caution** When setting a **Move Fixed** action with a speed of 0, you must also select the center square of the direction grid. If no direction square is selected at all, then the action is ignored!

Rooms

Our dragon and boss objects are all ready to go now, but in order to see them we need to put them into a level. Levels in Game Maker are made using *rooms*, and putting objects into a room defines where they will appear at the start of the game. However, not all objects need to be there at the start of the game, and they can be created on the fly as well (fireballs, for example). Let's create a new room.

Creating a new room resource:

1. Select **Create Room** from the **Resources** menu. A Room Properties form will appear (see Figure 2-14).

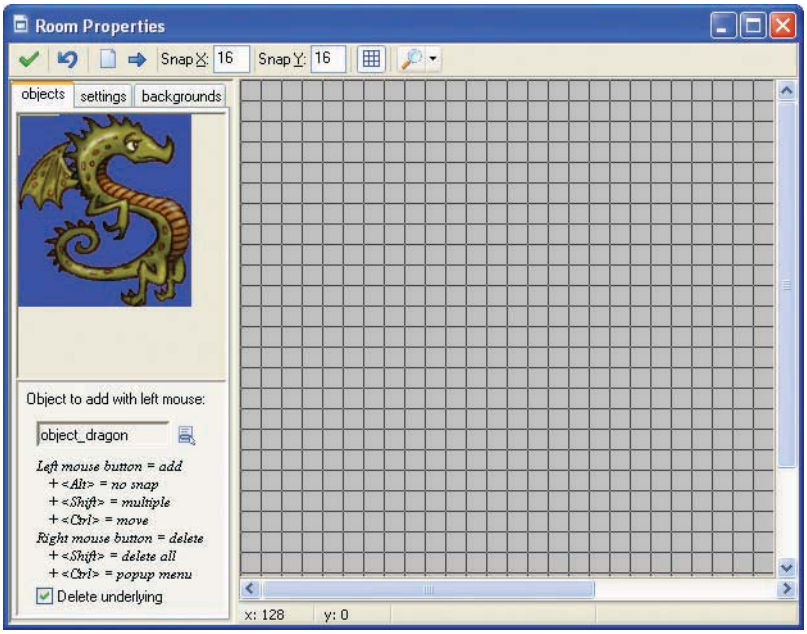

Figure 2-14. *The Room Properties form for a new room opens.*

■ **Note** If there are sliders along the edges of the room grid, then the window is not currently large enough to see the entire room. Maximize the Game Maker window and the Room Properties form to see more of the room, or use the sliders to scroll around the entire room.

2. Click the **settings** tab in the top left of the form.
3. Enter a name for the room in the **Name** field. Call this one `room_first`.
4. Enter a caption for the title bar of the window when the game is running. "Evil Clutches" seems appropriate for this game. The room settings should now look like Figure 2-15.

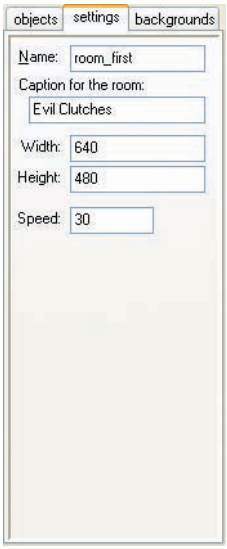

Figure 2-15. *Here's the **settings** tab of the Room Properties form, with the name and caption filled in.*

Now we can place our objects in the new room.

Adding a dragon and boss to the room:

1. Click the **objects** tab in the top left of the form. You should see that the dragon object is already selected as the object "to add with left mouse."
2. Click on the room grid with the left mouse button. An instance of the dragon object will be placed with its top-left corner at the point at which you click. The position you place the dragon becomes its starting position in the game, so put just one dragon close to the left boundary of the room area. If you add it in the wrong place, use the right mouse button to remove it again.
3. Click on the dragon's image on the **objects** tab (or on the image of the pop-up menu next to where it says `object_dragon`) and select the boss object from the menu that appears.
4. Place an instance of the boss close to the right edge of the room, but make sure that the whole of his sprite is completely inside the room—otherwise his events will not work correctly! The room should now look something like Figure 2-16.

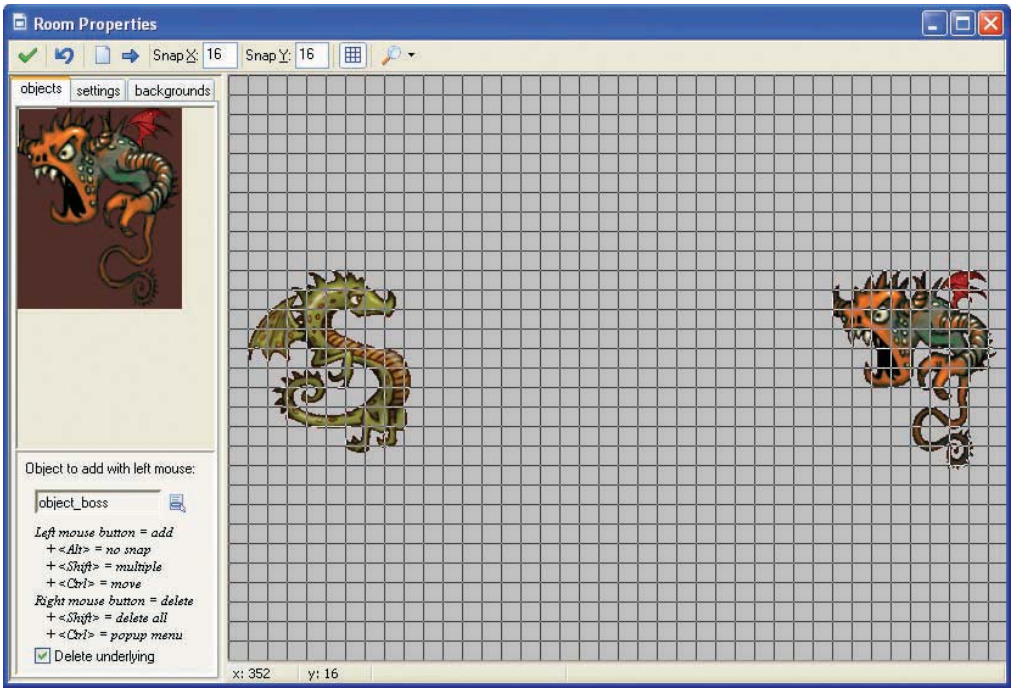

Figure 2-16. *The room with the dragon and the boss looks like this.*

Our very first version of the game is now ready. Click the green checkmark in the top-left corner of the form to close it and you can see the results of your labor . . .

Tip You can also click and hold the mouse button to move instances within a room.

Save and Run

It's always a good idea to save your work as often as possible—just in case your computer crashes. In case you haven't already worked it out for yourself, then the steps for this process are given here. However, in the future you'll have to remember to save your work regularly yourself! This works in the same way as most programs.

Saving your work and running the game:

1. Choose **Save** from the **File** menu (or click the disk icon).
2. The first time you save the game, you will be prompted for a location and filename in the normal way. Note that Game Maker files always end with the extension `.gm6`. Save this game in a place where you can easily find it again (on the desktop, for example).

3. To run the game, select **Run Normally** from the **Run** menu. After a brief pause, a game window should appear, like the one shown in Figure 2-17.

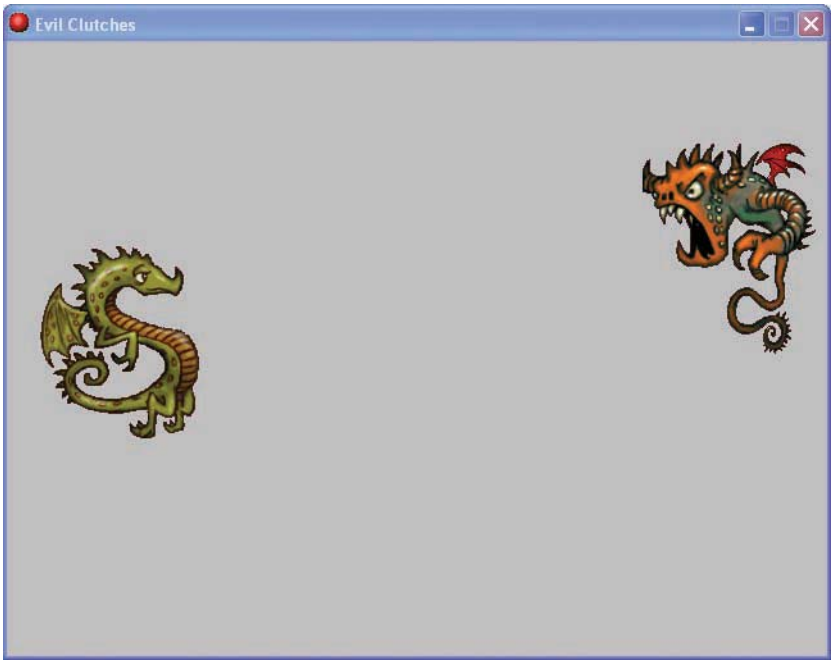

Figure 2-17. *Here's the first version of the Evil Clutches game in action.*

You should now be able to move the dragon up and down using the arrow keys, and the boss should float up and down by itself. If your game doesn't work in this way, then you might want to check through all your steps in the previous sections. You may also need to ensure that your game window is selected (by clicking on it with the mouse) before your keyboard input has any effect. All games we make in the book are stored on the CD in stages, and the current version of the game can be found on the CD in the file `Games/Chapter02/evil1.gm6`.

Although we now have a running game, it's not much fun to play yet as there are no goals or challenges. We'll spend the remainder of the chapter turning it into a playable game. Press Esc to stop the game.

■ **Tip** Pressing F4 while the game is running will maximize the game to fill the entire screen. Press F4 again to return to the windowed version.

Instances and Objects

So far we have two object resources in our game and two characters appearing on the screen. However, there is an important distinction to be made between object resources and *instances* of objects that appear on screen. It may seem odd, but now that we have made dragon and boss objects, we can put as many *instances* of dragons and bosses on the screen as we like. Try it—go back and place more dragons and bosses in the room. If you run the game, you will find that they all behave in exactly the same way as the original instances! (Don't forget to remove them again afterward using the right mouse button.) A good way to think of the relationship between objects and instances is to think of objects as jelly molds and instances as the jellies that you make with them. You only need one mold to make any number of jellies, yet the mold defines the appearance of all of them (see Figure 2-18). From now on we will talk about instances and objects in this way, so it is important that you appreciate the difference.

Figure 2-18. *Object resources are like jelly molds, and they can be used to create any number of object instances on the screen at once.*

Demons, Baby Dragons, and Fireballs

To create some challenges and goals, we're going to need to bring our remaining objects into the game. Let's start by giving the dragon the ability to breathe fireballs—as dragons often do!

The Fireball Object

To create the fireball object you'll need the fireball sprite. If you didn't get around to doing this earlier, then quickly flick back a few pages and add it in the way that was described in the "Sprites" section. You should remember the basic steps required to making a new object by now, but here they are one more time, just in case.

Creating the fireball object:

1. Select **Create Object** from the **Resources** menu.
2. Call the object `object_fireball`.
3. Select the fireball sprite.

We now need to think about how we want fireballs to behave. When the dragon creates a fireball, we want it to move across the screen toward the boss and get destroyed when it reaches the other side of the screen.

Adding the fireball object's events:

1. Click the **Add Event** button and choose the **Create** event.

2. Include the **Move Fixed** action in the **Actions** list. Select the right arrow to indicate the direction and set **Speed** to `32` (fireballs fly fast!).
3. Click the **Add Event** button again, select **Other** events, and pick **Outside room**.

4. Select the **main1** action tab and include the **Destroy Instance** action in the **Actions** list. In the action form that pops up, simply click **OK**. The fireball Object Properties form should now look like Figure 2-19.
5. Click **OK** to close the fireball Object Properties form.

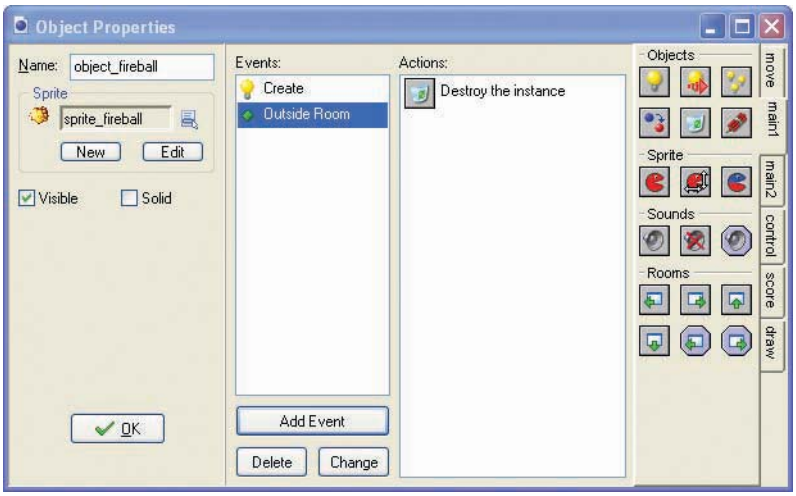

Figure 2-19. *The properties form for the fireball object should now look like this.*

Caution It is always a good idea to make sure that instances are deleted when they're not needed any more (when they go off the edge of the screen, for example). Even though you can't see them, Game Maker still has to spend time updating them, and too many instances will eventually slow down the program.

Now we need to tell the dragon object to create instances of the fireball object when the player presses the spacebar. We do this in a similar way to the events that make the dragon move, but this time using a **Key Press** event rather than a **Keyboard** event. **Keyboard** events happen as long as the player continues to hold down the key, but **Key Press** events happen only once when the key is first pressed. Using a **Keyboard** event for the fireballs would create a continuous stream of fireballs and make the game too easy, so that's why we're using **Key Press** instead.

Creating a Key Press event for the dragon object:

1. Double-click the dragon object in the resource list (not the dragon sprite). This will bring back the Object Properties form for the dragon object.
2. Click the **Add Event** button. Select the **Key Press** event and then choose **<Space>** from the pop-up menu.

3. Select the **main1** action tab and include the **Create Instance** action in the **Actions** list.
4. In the action form that appears, we need to specify which type of instance to create and where on the screen it should be created. Select the fireball object from the menu, enter a value of 100 into **X** and 10 into **Y**, and select the **Relative** checkbox. Figure 2-20 shows what the completed action form should look like.
5. Click **OK** to close the action form and click **OK** again to close the Object Properties form.

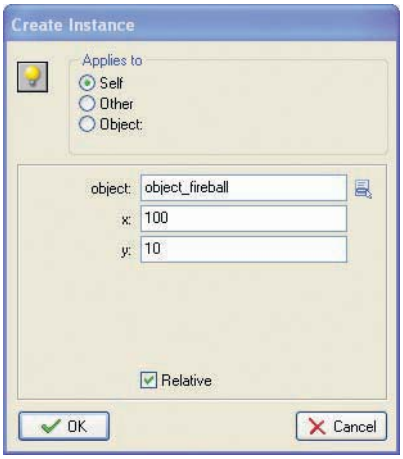

Figure 2-20. *Note that we checked the* ***Relative*** *property to make the fireball appear relative to the position of the dragon.*

The x and y values you just entered are screen coordinates, which are used to indicate positions on the game screen. Screen coordinates are measured in pixels (the tiny squares that make up a monitor display), with x values indicating the number of pixels horizontally, and y values indicating the number of pixels vertically.

We need to select the **Relative** option because the fireball needs to be created on the screen in front of the dragon, in other words, *relative* to the dragon's position. However, the dragon's position is measured from the top-left corner of its sprite—just above its wings—and this would be a crazy place for the fireball to appear! Giving an x-coordinate of 100 moves the fireball across 100 pixels to the right (to just above its head) and a y-coordinate of 10 brings it 10 pixels down. This creates the fireball right in front of the dragon's mouth and exactly where we need it (see Figure 2-21). Test the game now to check that you can use the spacebar to shoot fireballs, and that they appear in the correct position.

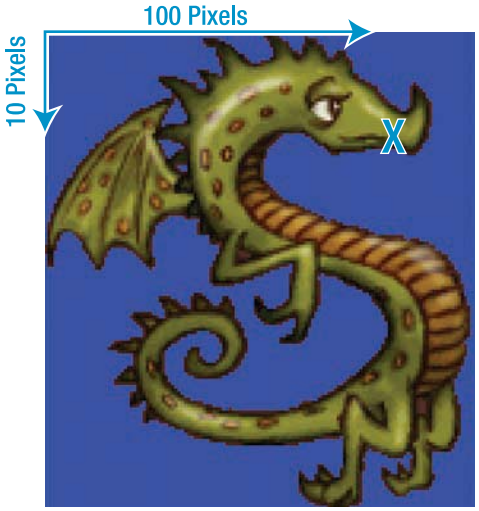

Figure 2-21. *The fireball needs to appear from the dragon's mouth, which is 100 pixels across and 10 pixels down from the origin of the dragon's sprite.*

The Demon Object

The demon object will work in the same way as the fireball, except that demons fly from right to left and are created by the boss. Also, to make demons a bit more interesting, we'll start some moving diagonally as well as horizontally. Those that head diagonally for the top or bottom of the screen will need to reverse their vertical direction when they intersect the boundary—like the boss object does. We'll also need to destroy demons when they go outside the room, like the fireball. Next we provide the steps you need to do all of this; notice that we've started to shorten the steps that you should be familiar with by now.

Creating the demon object:

1. Create a new object called `object_demon` and give it the demon sprite.

2. Add a **Create** event and include the **Move Fixed** action.
3. Select all three left-pointing direction arrows and set **Speed** to `12`. Selecting more than one direction causes Game Maker to randomly choose between them when an instance is created. The action form should now look like Figure 2-22.

4. Add an **Intersect boundary** event (in the **Other** events) and include the **Reverse Vertical** action in it.

5. Add an **Outside room** event (also in the **Other** events) and include a **Destroy Instance** action in it.

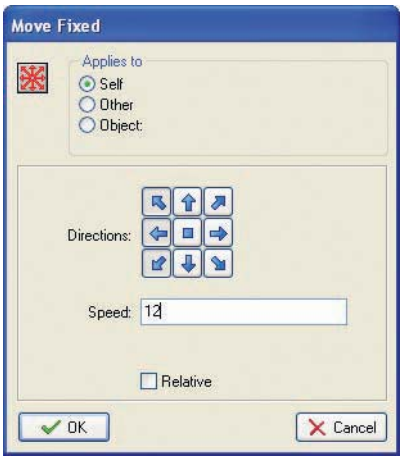

Figure 2-22. *In the move action for the demon, note that all three arrows to the left are pressed, so a random direction out of the three is selected for each demon created.*

The demons will now bounce back and forth between the top and the bottom of the screen, but we also need them to react to collisions with other instances. For this we use a *collision event*, which happens when two sprites of different objects overlap on the screen. The first collision event we need is for when a demon collides with a fireball. This event should destroy the demon, and reward the player by increasing their score. There are a number of different actions dealing with scores, health, and lives in the **score** actions tab. As soon as the score changes, it will automatically be displayed in the game window caption.

Adding an event to the demon object for colliding with the fireball:

1. Click the **Add Event** button, choose the **Collision** event, and select the fireball object from the pop-up menu.

2. Include the **Destroy Instance** action from the **main1** action tab.

3. Also include a **Set Score** action from the **score** tab. This should automatically appear below the **Destroy Instance** action in the **Actions** list. Lists of actions like this are carried out one after another, starting from the top of the list and working down.

4. Enter a value of 100 in the **Set Score** action form, and click the **Relative** property. This property makes the action set the score *relative* to the current score, so 100 will be added to the score rather than setting the score to 100. See Figure 2-23.

If a demon collides with the dragon, then the game is over. When this happens, we want to bring up a high-score table and (when appropriate) let the player enter their name. After showing the high-score table, we want to restart the game. Conveniently, Game Maker provides a **Show Highscore** event that handles most of this automatically.

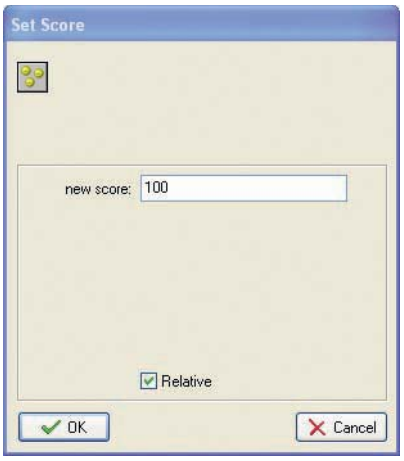

Figure 2-23. *We add 100 to the score by setting the* ***Relative*** *property.*

Adding an event to the demon object for colliding with the dragon:

1. Add a **Collision** event for colliding with the dragon object.

2. Include a **Show Highscore** action from the **score** tab.
3. Click **OK** to keep the default settings for this action's properties.

4. Also include a **Restart Game** action from the **main2** tab. This action has no properties.
5. The object properties form for the demon should now look like Figure 2-24. Check that you have included all the demon object's events. We're done with this object for now, so click **OK**.

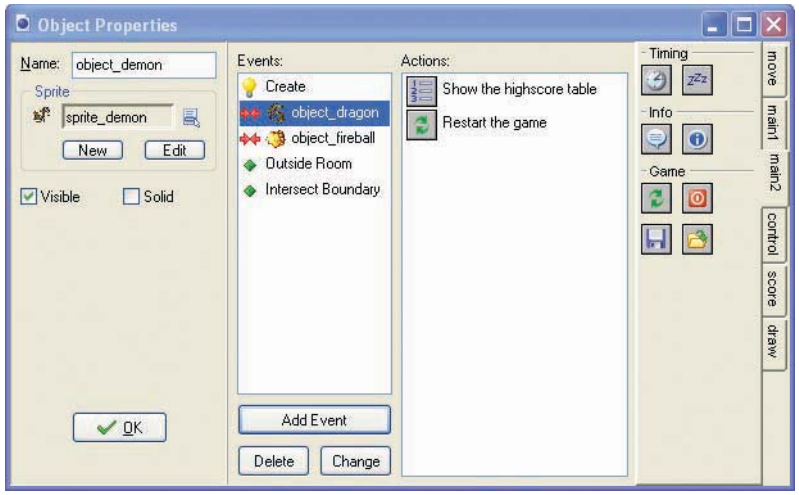

Figure 2-24. *The Object Properties form for the demon object now looks like this.*

Summoning Demons

That's it for the demon, but we still need the boss to create instances of the demon in the first place. However, we don't want the demons to appear at regular intervals because this would make the game too easy. Instead, we want there to be a random chance that a demon is created at each "*step*" of the game. A *step* is essentially just a short period of time in which everything on the screen moves a very small distance. There are normally 30 steps in every second, so we only need there to be a very small chance that a demon is created in each step. We achieve this by using a **Test Chance** action, which acts like throwing a die with many sides (see Figure 2-25). In each step we throw the die, but only one side will trigger the chance action and create a demon. In this way, we create a steady, but unpredictable, flow of demons.

Figure 2-25. *The more sides a die has, the less often Game Maker will throw the one side that triggers the* ***Test Chance*** *action.*

Adding a step event to the boss object:

1. Double-click the boss object in the resource list to bring back its Object Properties form.
2. Click the **Add Event** button, select the **Step** event, and choose **Step** again from the pop-up menu.
3. Include the **Test Chance** action from the **control** tab. Set the sides of the die to 50 in the action's properties.
4. Also include the **Create Instance** action in the **Actions** list for this event. Set the properties to create a demon object and select the **Relative** option, so that the demon is created relative to the boss's position. The event should now look like Figure 2-26.

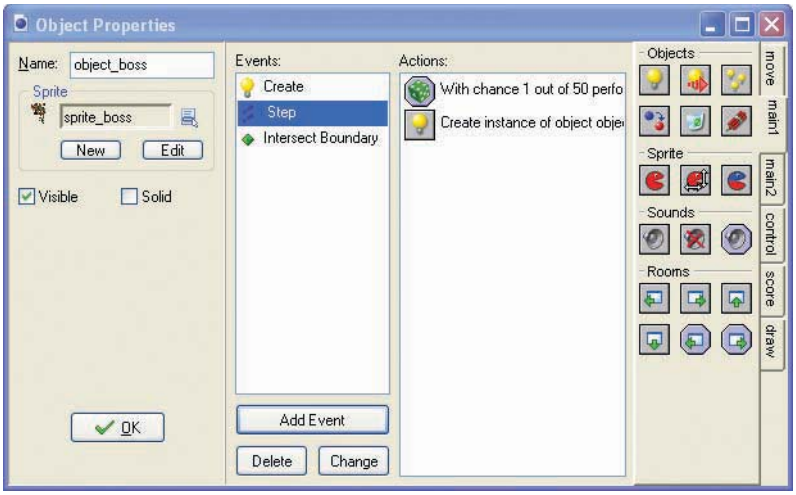

Figure 2-26. *In this step event, demons are randomly created.*

The **Test Chance** action is an example of a *conditional action*. Conditional actions control the action that immediately follows them so that it is only performed if some condition is met. So in this case the **Create Instance** action is only performed if the **Test Chance** action rolls a 1 using a 50-sided die—otherwise it is skipped.

Click **OK**, save your work, and run the game to test it. Demons should now be appearing, and you should be able to shoot them with your fireballs to rack up your score in the window caption. When you eventually get hit by a demon, the high-score table will be displayed and the game restarts. How long can you survive?

The Baby Dragon Object

We now have a game with two goals: shooting demons and staying alive. However, it's still not much fun to play as it's far too easy to provide any real challenge. To increase the challenge, we're going to occasionally throw in a baby dragon along with the demons. If the player shoots a baby dragon, they will lose 300 points, but if they rescue one they will gain 500 points. This will mean that the player will have to be much more careful about when they shoot, thereby increasing the challenge of the game.

Creating a new baby dragon object and its events:

1. Create a new object called `object_baby`, and give it the baby dragon sprite.

2. Add a **Create** event for the object and include a **Move Fixed** action in it. Set it to move left with a **Speed** of `8` (slower than the demons to make life harder).

3. Add an **Outside room** event (in **Other** events) and include a **Destroy Instance** action from the **main1** tab.

4. Add a **Collision** event with the fireball object and include a **Destroy Instance** action in that as well.

5. Also include a **Set Score** action in the collision event with a value of `-300` and the **Relative** property selected. This will subtract 300 from the player's score.

6. Add a **Collision** event with the dragon object and include the **Destroy Instance** action in it.

7. Also include the **Set Score** action with a value of `500` and the **Relative** property selected. This will add 500 to the player's score. The baby dragon object should now look like Figure 2-27.

8. Click **OK** to close the properties form.

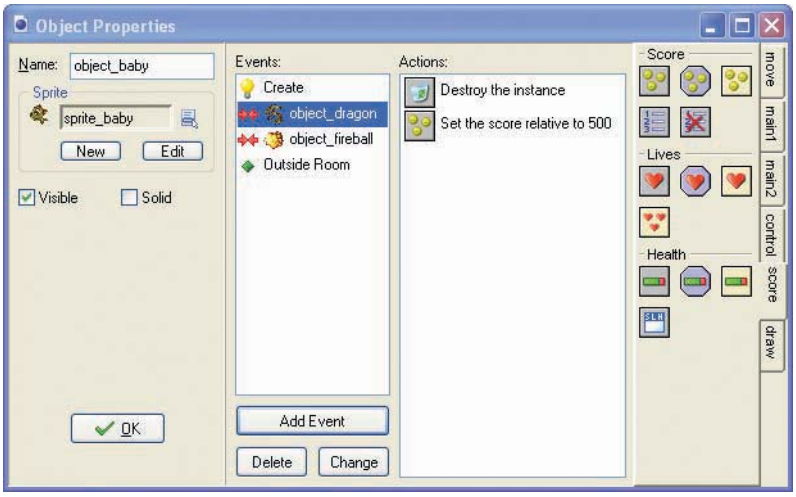

Figure 2-27. *The Object Properties form for the baby dragon object looks like this.*

Now we need to make the boss randomly release baby dragons as well as demons. This is exactly the same as for the demons except we will use a value of 100 for the die so that they are created less often.

Editing the boss object to randomly create baby dragons:

1. Reopen the Object Properties form for the boss object.

2. Click on the existing **Step** event to select it and view its actions.

3. Include another **Test Chance** action in the **Step** event. Set the sides of the die to be `100` in the action's properties.

4. Include the **Create Instance** action below the new **Test Chance** action in the **Actions** list. Set the properties to create a baby object and select the **Relative** option.

That completes the second phase of our game! All the gameplay elements are now in place. Save the game and carefully test it to make sure it works correctly. You'll also find the current version of the game on the CD in the file `Games/Chapter02/evil2.gm6`.

Backgrounds and Sounds

In this section we'll finish off the look and feel of our game by adding background graphics, sound effects, and music. As you'll see, these finishing touches have quite a dramatic effect on how professional the game seems.

A Background Image

The first improvement we'll make is to add a background to the room. Backgrounds are another type of resource, like sprites, rooms, and objects. We've created an image that is exactly the same size as the game window (640×480 pixels). This needs to be loaded into a new background resource, which can then be assigned to a room.

Creating a new background resource and assigning it to a room:

1. Select **Create Background** from the **Resources** menu.
2. Call the background `background_cave`, and click the **Load Background** button. Select the `Background.bmp` image from the `Resources/Chapter02` folder on the CD. The Background Properties form should now look like Figure 2-28.
3. Click **OK** to close the Background Properties form.
4. Reopen the properties form for the room by double-clicking on it.
5. Select the **backgrounds** tab in the Room Properties form. Click the menu icon to the right of where it says <no background> and select the new background from the pop-up menu. The Room Properties form now looks like Figure 2-29.
6. Close the Room Properties form by clicking the green checkmark in the top-left corner of the form.

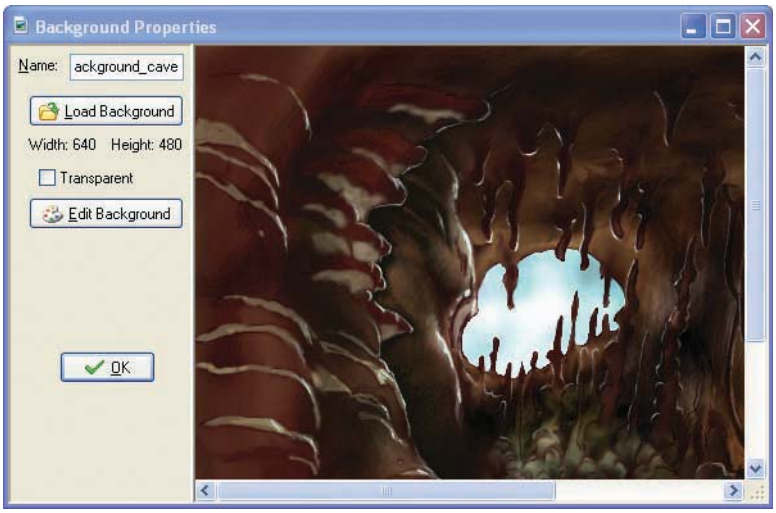

Figure 2-28. *The Background Properties form allows you to load and edit backgrounds.*

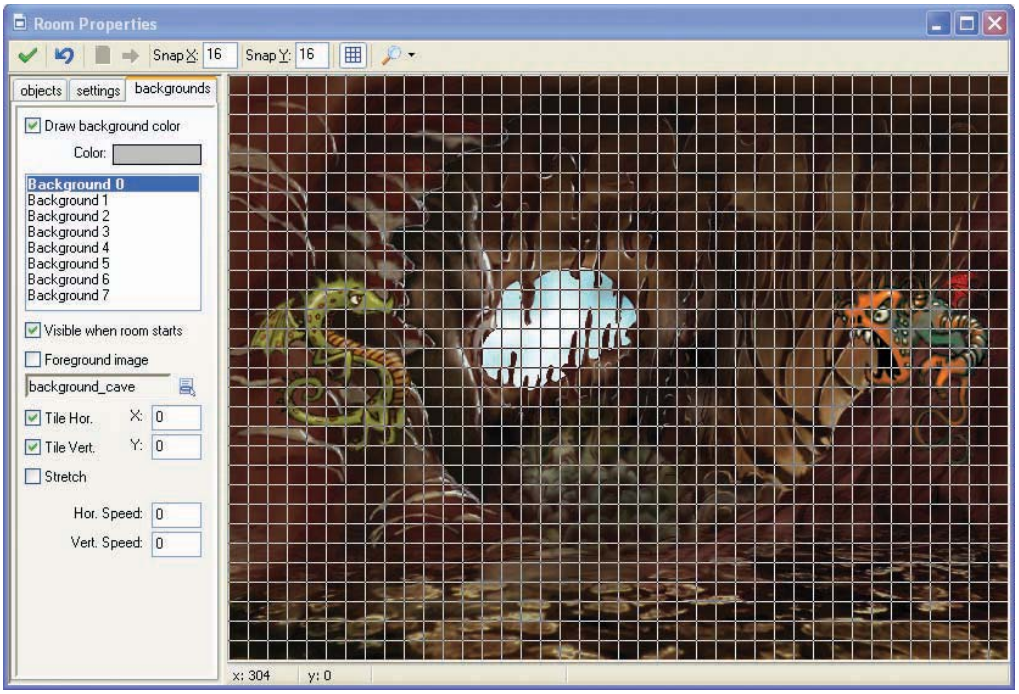

Figure 2-29. *The room looks a lot more atmospheric once you add the background to the room.*

Background Music

The next step is to add some atmospheric music. *Sounds* are another kind of Game Maker resource for including both sound effects and music. We need to create a sound resource for the music and then set up an action to start the music playing. We'll include this action in the **Create** event of the boss object so that it starts playing at the beginning of the game, but it would work just as well in the dragon object too.

Creating a music sound resource and playing it in the boss object:

1. Select **Create Sound** from the **Resources** menu and call it `sound_music`.
2. In the properties form that appears, click **Load Sound** and select the `Music.mp3` file from `Resources/Chapter02` on the CD. The Sound Properties form should now look like Figure 2-30.
3. Close the Sound Properties form by clicking **OK**.
4. Reopen the Object Properties form for the boss object.
5. Click the existing **Create** event to select it and view its actions.

6. Include a **Play Sound** action (**main1** tab) in the **Create** event.

7. In the action properties, select the music sound and set the **Loop** property to true. This makes the music loop back to the start when it finishes. The sound action form should then look like Figure 2-31.

8. Click **OK** to close the action, and click **OK** again to close the boss object.

Figure 2-30. *The Sound Properties form allows you to load, preview, and save sound files.*

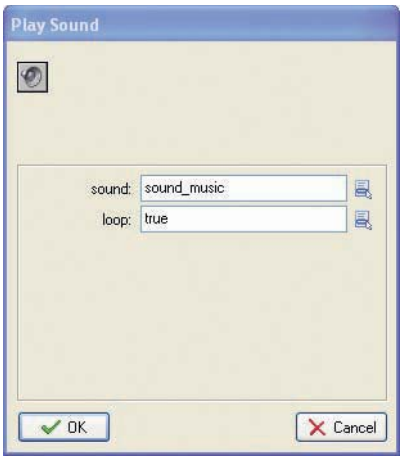

Figure 2-31. *This sound action loops the background music.*

Sound Effects

Adding sound effects is another way to enhance the atmosphere of a game, but they also help to inform the player about their actions. For now, we'll just add two sound effects to our game: one for shooting a demon and one for shooting a baby dragon. The baby's sound effect will be much higher-pitched and cuter than the demon's so that the player instantly knows they have done something wrong.

Creating and playing sound effects for shooting babies and demons:

1. Create a new sound resource called `sound_demon`.

2. Load the `Demon.wav` file from `Resources/Chapter02` on the CD.

3. Close the Sound Properties form.

4. Reopen the Object Properties form for the demon object and select the existing **Collision** event with the fireball object.

5. Include a **Play Sound** action in the collision event and select the new sound. Leave the **Loop** property set to false.

6. Close the action form and demon Object Properties form.

7. Repeat the previous steps to create a sound resource for `Baby.wav`. Include an action to play it in the baby dragon object's collision event with the fireball.

Congratulations

Congratulations on completing your very first game using Game Maker! If you need it, then you'll also find the final version of the game in the file `Games/Chapter02/evil3.gm6`, on the CD. When you've finished a game, you can turn it into an executable by choosing **Create Executable** from the **File** menu. Executables don't need Game Maker to run, so it's easy to give them to your friends or put them on a website.

Now that you're a bit more familiar with Game Maker, why not try making some changes to the game to see what effects they have? You could add new objects to the game—there's an image for an "evil baby" in the resources directory that you can use. Perhaps these could be demons in disguise? Also try changing the movement speeds of the different objects. This can have a big impact on the difficulty of the game, as can changing the number of sides on the dice in the **Test Chance** actions. Balancing the settings for all these parameters is one of a game designer's most important jobs, and we'll talk more about this in Chapter 11.

This chapter has introduced you to the basic elements of Game Maker. We've looked at different kinds of game resources and seen how events and actions are used to control the behavior of objects. However, we've only just scratched the surface; there is still much more to discover about Game Maker and lots of even better games to make. Our journey continues in the next chapter with a trip to a moon or two as we learn more about events and actions by playing with spaceships.

PART 2

Action Games

There aren't many jobs where you try to put your customers into dangerous situations, but asteroid fields are just occupational hazards in this line of work!

CHAPTER 3

More Actions: A Galaxy of Possibilities

We hope you enjoyed making Evil Clutches and that it gave you a sense of how easy Game Maker is to use. However, you can achieve so much with a bit more knowledge, so let's move on to our second project and do something a little more adventurous.

Designing the Game: Galactic Mail

As before, it helps to set out a brief description of the game we want to create. We'll call this game *Galactic Mail* because it's about delivering mail in space. Here's the design:

You play an intergalactic mail carrier who must deliver mail to a number of inhabited moons. He must safely steer a course from moon to moon while avoiding dangerous asteroids. The mail carrier is paid for each delivery he makes, but pay is deducted for time spent hanging around on moons. This adds pressure to the difficult task of orienting his rickety, old rocket, which he cannot steer very well in space.

When the rocket is on a moon, the arrow keys will rotate it to allow the launch direction to be set. The spacebar will launch the rocket, and the moon will be removed from the screen to show that its mail has been delivered. In flight, the rocket will keep moving in the direction it is pointing in, with only a limited amount of control over its steering using the arrow keys. When things move outside the playing area, they reappear on the other side to give the impression of a continuous world. The player will gain points for delivering mail, but points will be deducted while waiting on a moon. This will encourage the player to move as quickly as possible from moon to moon. There will be different levels, with more asteroids to avoid. The game is over if the rocket is hit by an asteroid, and a high-score table will be displayed. Figure 3-1 shows an impression of what the final game will look like.

This description makes it possible to pick out all the various elements needed to create the game, namely moons, asteroids, and a rocket. For reasons that you will see later, we'll actually use two different moon objects (for a normal moon and an occupied moon) and two different rocket objects (for a "landed rocket" and a "flying rocket"). All the resources for this game can be found in the `Resources/Chapter03` folder on the CD.

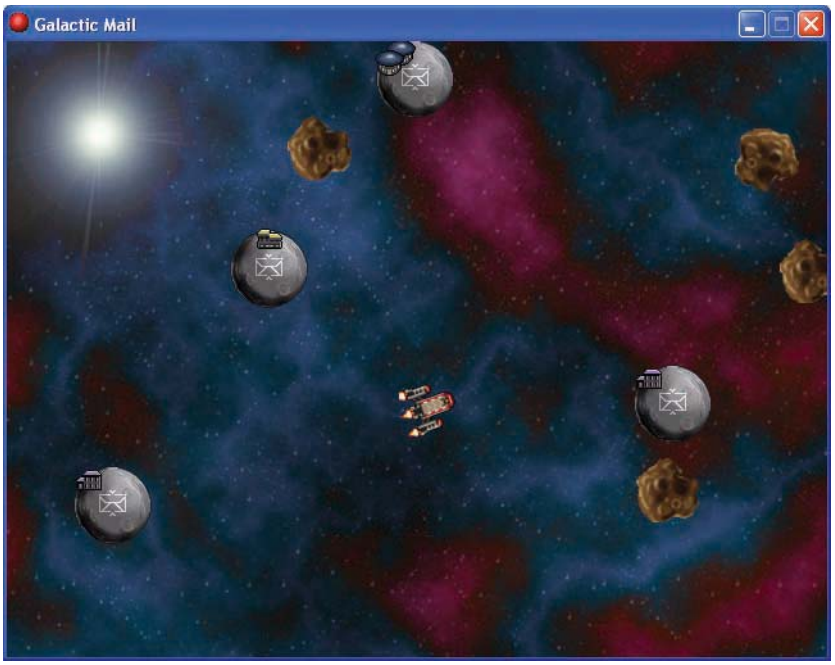

Figure 3-1. *The Galactic Mail game features moons, asteroids, and a rocket ship.*

Sprites and Sounds

Let's begin by adding all the sprites to our game. In the previous chapter, we saw that sprites provide images for each element of the game. In this chapter, we'll use some extra abilities of sprites; however, before we can do this, you must set Game Maker into Advanced mode.

Setting Game Maker into Advanced mode:

1. If you are working on a game, you must save the game before switching modes.
2. Click the **File** menu and look for an item called **Advanced Mode**. If there is a check-mark in front of it, then you are already in Advanced mode. Otherwise, click that menu item to select it, and the main window should change to look like the one in Figure 3-2.

To make things simple, we'll leave Game Maker in Advanced mode for the remainder of the book, even though some of the options will only be used in the final chapters. Now we're going to start a new, empty game.

■ **Note** To start a new game, choose **New** from the **File** menu. If you are already editing a game that has had changes made to it, you will be asked whether you want to save these changes.

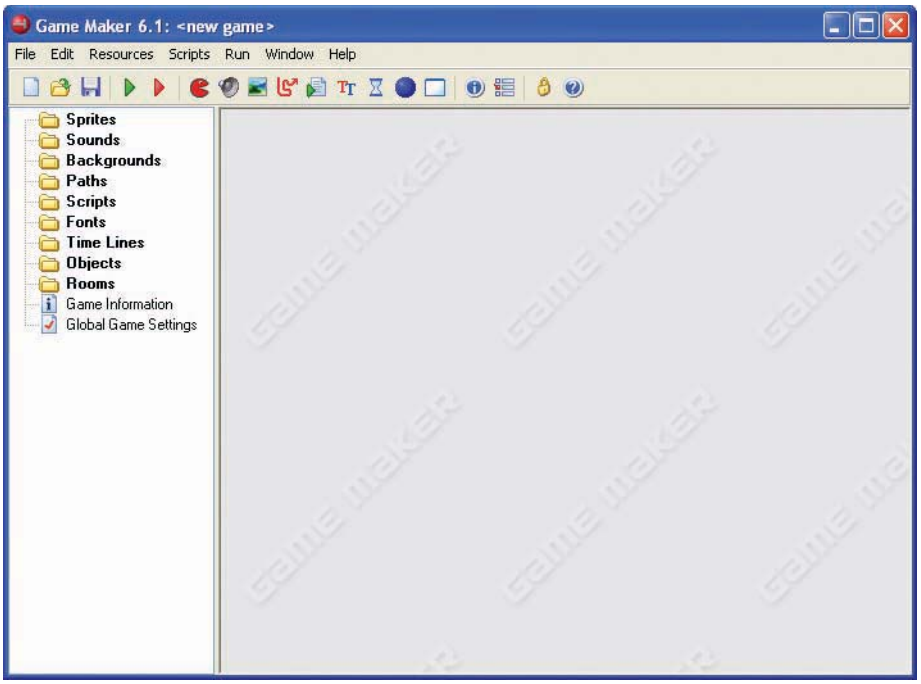

Figure 3-2. *In the main window of Game Maker in Advanced mode, there are a number of additional resources on the left and an additional menu.*

Our first step is to create all the sprites we need for the game. This works in the same way as in the previous chapter, but this time we must complete a couple of additional steps. Each sprite in Game Maker has its own *origin*, which helps to control the exact position in which it appears on the screen. By default, the origin of a sprite is set to be located at the top-left corner of the image. This means that when you move objects around in the game, it is as if you were holding them by their top-left corner. However, because the rockets in Galactic Mail need to sit in the center of the moons, it will be easier if we change the origin of all our sprites to be central.

Creating new sprite resources for the game:

1. From the **Resources** menu, choose **Create Sprite**. The Sprite Properties form with additional Advanced mode options will appear, like the one shown in Figure 3-3.
2. Click in the **Name** field and give the sprite a name. You should call this one `sprite_moon`.
3. Click the **Load Sprite** button. Select `Moon.gif` from the `Resources/Chapter03` folder on the CD.

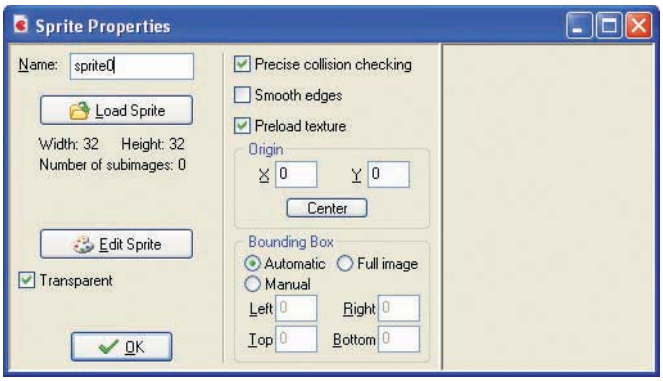

Figure 3-3. *This Sprite Properties form shows the advanced options.*

4. The controls for setting the origin are halfway down the second column of the form. Click the **Center** button to move the origin to the middle of the sprite. You should now see a cross in the middle of the sprite's image indicating the position of the origin. You can also change the origin by clicking on the sprite image with the mouse or typing in the **X** and **Y** values directly.

5. Enable the **Smooth edges** option by clicking on the box next to it. This will make the edges of the sprite look less jagged during the game by making them slightly transparent.

6. Click **OK** to close the form.

7. Now create asteroid and explosion sprites in the same way using `Asteroid.gif` and `Explosion.gif` (remember to center their origins too).

8. We'll need two sprites for the rocket: one for when it has landed on a moon and one for when it is flying through space. Create one sprite called `sprite_landed` using `Landed.gif` and another called `sprite_flying` using `Flying.gif`. Center the origins of these two sprites as before.

Before closing the Sprite Properties form for this last sprite, click the **Edit Sprite** button. A form will appear like the one shown in Figure 3-4. If you scroll down the images contained in this sprite, you'll see that it contains an animation of the rocket turning about a full circle. There are 72 different images at slightly different orientations, making up a complete turn of 360 degrees. We'll use these images to pick the correct appearance for the rocket as it rotates in the game. We can use the Sprite Editor to change the sprite in many ways, but for now simply close it by clicking the button with the green checkmark in the top left of the window.

Your game should now have five different sprites. Next let's add some sound effects and background music so that they are all ready to use later on.

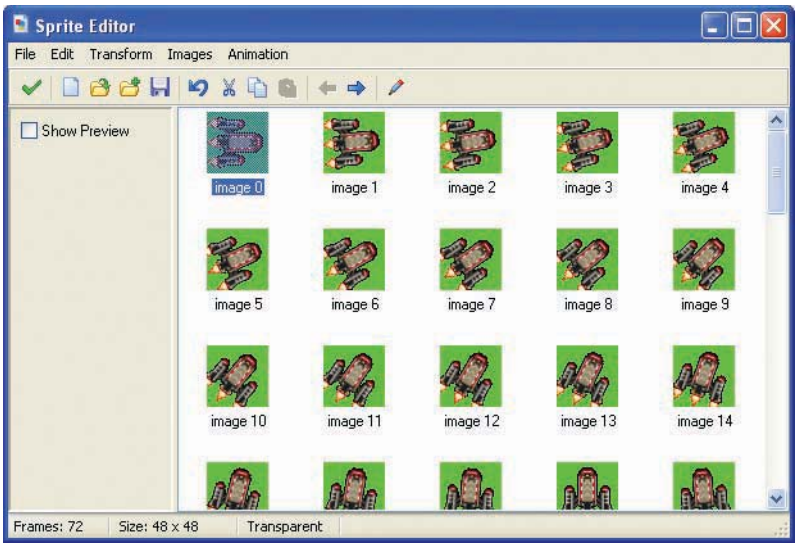

Figure 3-4. *The Sprite Editor shows all the images of the rocket.*

Creating new sound resources for the game:

1. Select **Create Sound** from the **Resources** menu. Note that the Sound Properties form now has additional Advanced mode options, but we don't need to worry about them for now (some of these are only available in the registered version of Game Maker).
2. Call the sound `sound_explosion` and click **Load Sound**. Select the `Explosion.wav` file from `Resources/Chapter03` on the CD.
3. Close the form by clicking **OK**.
4. Now create the `sound_bonus` and `music_background` sounds in the same way using the `Bonus.wav` and `Music.mp3` files.

Adding all these resources at the start will make it easier to drop them into the game as we are going along—so let's get started on some action.

Moons and Asteroids

Both moons and asteroids will fly around the screen in straight lines, jumping to the opposite side of the room when they go off the edge of the screen. In Game Maker this is called *wrapping*, and it is done using the **Wrap Screen** action.

Creating the moon object:

1. From the **Resources** menu, choose **Create Object**. The Advanced mode Object Properties form has additional options and actions too (see Figure 3-5).
2. Call the object `object_moon` and give it the moon sprite.

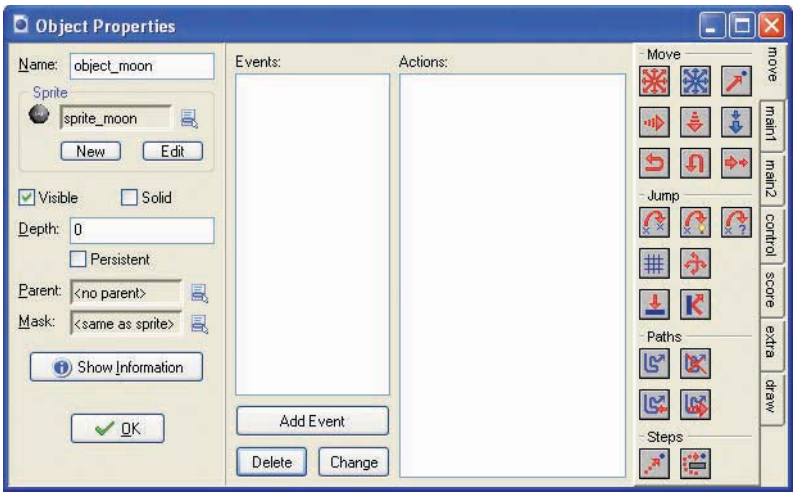

Figure 3-5. *The Object Properties form for the moon object looks like this.*

When a moon is created, we want it to start moving in a completely random direction.

Adding a create event to the moon object:

1. Click the **Add Event** button and choose the **Create** event.

2. Include the **Move Free** action in the **Actions** list for this event.

3. This action form requires a direction and a speed. Enter a **Speed** of `4` and type `random(360)` in the **Direction** property. This indicates a random direction between 0 and 360 degrees. The form should now look like Figure 3-6.

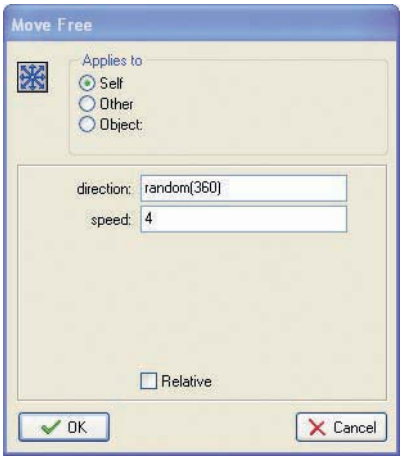

Figure 3-6. *Using the random command in a* ***Move Free*** *action will make the moons start moving in a random direction.*

We also need to make sure that when the moon goes off the edge of the room, it reappears at the other side.

Including a wrap action for the moon object:

1. Click the **Add Event** button, choose the **Other** events, and select **Outside Room** from the pop-up menu.

2. Include the **Wrap Screen** action in the **Actions** list.
3. In the form that appears, you should indicate that wrapping should occur in both directions (top to bottom and left to right). Now the form should look like Figure 3-7.
4. The moon object is now ready to go, so you can close the Object Properties form by clicking **OK**.

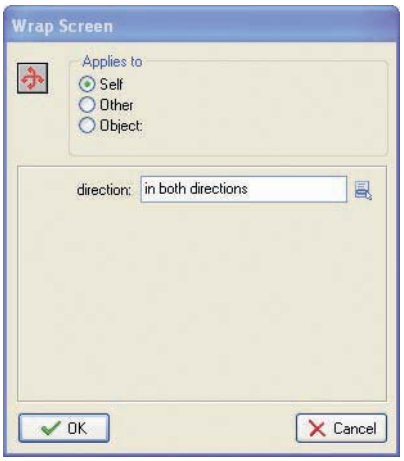

Figure 3-7. *The **Wrap Screen** action properties form looks like this.*

The asteroid object can be created in exactly the same way as the moon earlier. However, to keep things neat, we want to make sure that asteroids appear behind other objects when they cross paths with them on the screen. Instances of objects are usually drawn in the order in which they are created, so it is hard to be sure whether one type of object will appear in front of another. However, you can change this by setting an object's *depth* value. Instances with a smaller depth are drawn on top of instances with a larger depth, and so appear in front of them. All objects have a default depth of 0, so to make sure the asteroids appear behind other objects we simply give them a depth greater than 0.

Creating the asteroid object:

1. Create a new object called `object_asteroid` and give it the asteroid sprite.
2. On the left-hand side there is a text field labeled **Depth**. Enter `10` in this field to change the depth of the object from 0 to 10.

3. Add the **Create** event and include the **Move Free** action in the **Actions** list. Type `random(360)` in the **Direction** property and enter a **Speed** of `4`.

4. Add the **Other, Outside Room** event and include the **Wrap Screen** action in the **Actions** list (indicate wrapping in both directions).

Note From now on we will use commas in event names, such as **Other, Outside Room** to show the two stages involved in selecting the event.

---

5. The Object Properties form should now look like Figure 3-8. Click **OK** to close the form.

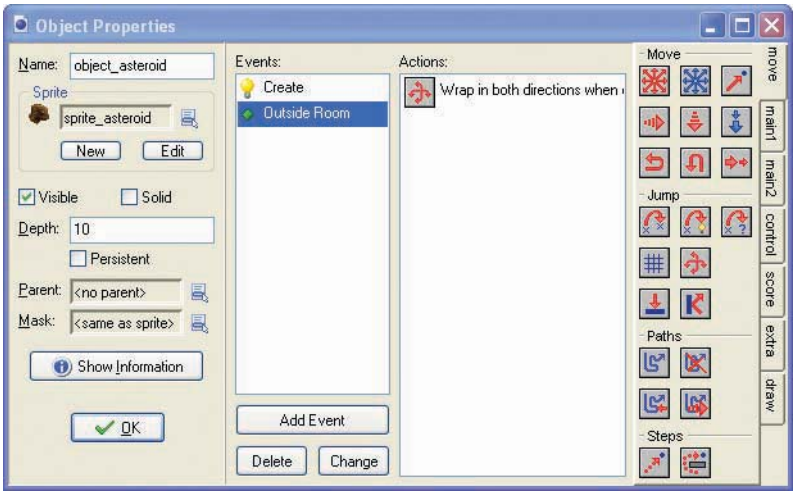

Figure 3-8. *We've set the depth for the asteroid object.*

Now would seem like a good time to check that everything has gone according to plan so far. However, before we can do that we must create a room with some instances of moons and asteroids in it.

Creating a room with moon and asteroid instances:

1. Select **Create Background** from the **Resources** menu.
2. Call the background `background_main`, and click the **Load Background** button. Select the `Background.bmp` image from the `Resources/Chapter03` folder on the CD.
3. Click **OK** to close the Background Properties form.
4. Select **Create Room** from the **Resources** menu. If the whole room isn't visible, then enlarge the window.

5. Select the **settings** tab and call the room `room_first`. Provide an appropriate caption for the room (for example "Galactic Mail").
6. Select the **backgrounds** tab. Click the menu icon to the right of where it says <no background> and select the background from the pop-up menu.
7. Select the **objects** tab and place a number of asteroids and moons in the room. (Remember that you can choose the object to place by clicking where it says "Object to add with left mouse"). The Room Properties form should now look like Figure 3-9.
8. Close the Room Properties form by clicking the green checkmark in the top-left corner.

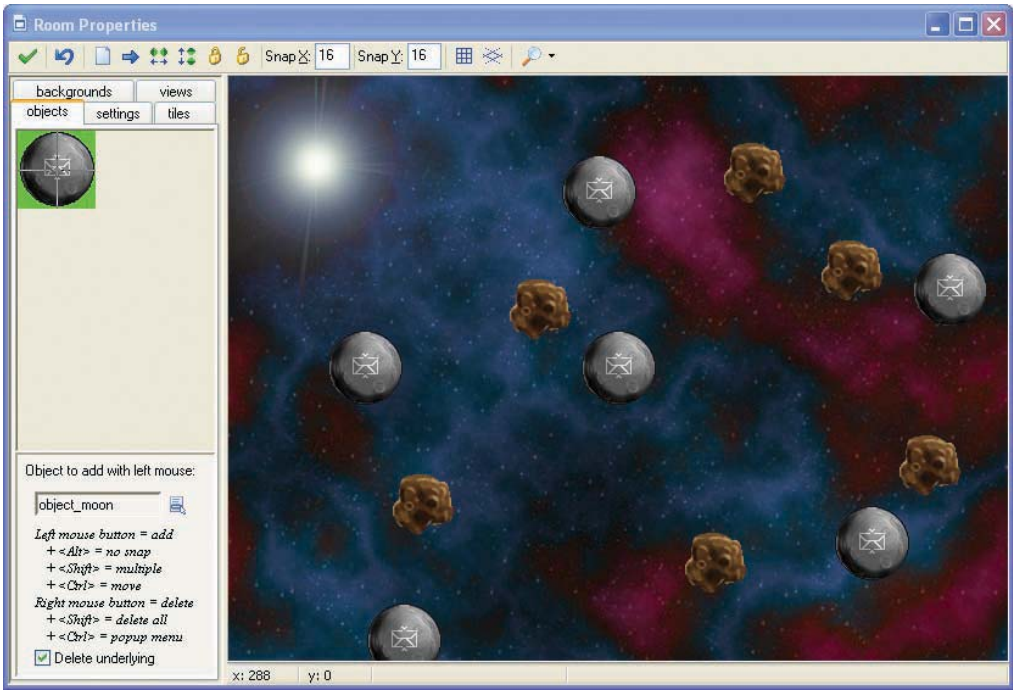

Figure 3-9. *Here's our first room.*

That should give us something to look at, so let's give it a try.

Saving and running the game:

1. Choose **Save** from the **File** menu (or click the disk icon). Save the game somewhere where you can easily find it again (the desktop, for example).
2. Select **Run normally** from the **Run** menu. If all goes well, the game should then appear in a new window.

Before continuing, double-check that everything is working the way it's supposed to. Are the moons and asteroids moving in different random directions? Do they reappear on the

other side of the screen when they leave the room? Do the asteroids always pass behind the moons? If any of these are not working, check that you have followed the instructions correctly. Alternatively, you can load the current version from the file `Games/Chapter03/galactic1.gm6` on the CD.

Flying Around

This isn't a very interactive experience yet, so let's introduce some gameplay by bringing the rocket into the game. We've already mentioned that we'll make two rocket objects, but let's stop to consider why this is necessary. Our rocket has two different ways of behaving: sitting on top of a moving moon with full control over the ship's direction, and flying through space with only limited control. Having two ways of controlling one object would involve a complicated set of events and actions, but if we separate these behaviors into two different objects, then it becomes quite simple. Provided that both objects look the same, the player will never notice that their ship is actually changing from being a "flying rocket" object to a "landed rocket" object at different stages of the game.

We also need two moon objects, as we want the landed rocket object to follow the path of one particular moon around (the one it has landed on). Making it into a separate object will allow us to single it out from the others in this way. As this second moon object will be almost the same as the normal moon, we can take a shortcut and make a copy of the existing moon object.

Creating the special moon object:

1. Right-click the moon object in the resource list, and select **Duplicate** from the pop-up menu. A copy of the moon object will be added to the resource list and its properties form is displayed.
2. Change the name to `object_specialmoon`. It is important that you use this exact name (including the underscore) as we will use this to identify this object later on.
3. Set the **Depth** of this object to `-5`. This will guarantee that instances of this moon are always in front of the other moons as it is lower than 0.

4. We will also make this moon responsible for starting the background music at the beginning of the game. Add an **Other**, **Game start** event and include a **Play Sound** action in it (**main1** tab). Select the background music sound and set **Loop** to true so that the music plays continuously.
5. Click **OK** to close the properties form.

Now open the first room and add a single instance of this new special moon to the level. Run the game and the music should play. (You won't notice any other difference because the special moon should look and behave exactly like the other moons.)

Now we can make our two rocket objects. We'll begin with the landed rocket, which needs to sit on the special moon object until the player decides to blast off. We'll use a **Jump Position** action to make it follow the special moon's position as it moves around the screen.

Creating the landed rocket object:

1. Create a new object called `object_landed` and give it the landed rocket sprite. Set the **Depth** to `-10` so that it appears in front of the moons and looks like it's sitting on the surface of the special moon.

2. Add a **Step, End Step** event to the new object. An **End Step** allows actions to be performed immediately before instances are drawn at their new position on the screen. Therefore, we can use this event to find out where the special moon has been moved to and place the rocket at the same location—just before both of them are drawn.

■ **Note** A **Step** is a short period of time in which everything on the screen moves a very small distance. Game Maker normally takes 30 steps every second, but you can change this by altering the **Speed** in the **settings** tab for each room.

3. Include the **Jump Position** action in the **Actions** list for this event. This action allows us to move an object to the coordinates of any position on the screen. Type `object_specialmoon.x` into the **X** value and `object_specialmoon.y` into the **Y** value. These indicate the `x` and `y` positions of the special moon. Make sure that you type the names carefully, including underscores and dots (i.e., periods or full stops) in the correct positions. The action should now look like Figure 3-10.

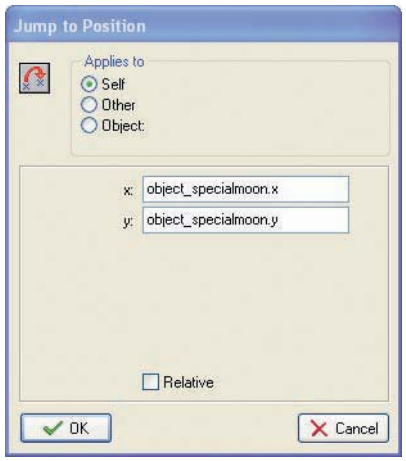

Figure 3-10. *We set the rocket to jump to the x and y positions of the special moon, so that it will follow this moon around.*

4. You might want to test the game now. Place one instance of the rocket at a random position in the room and run the game. The rocket should jump to the position of the special moon and stay on top of it as it moves around.

When you run the game, you will also notice that the rocket continually spins around without any user input. This is because the rocket sprite contains an animation showing the rocket rotating through 360 degrees. By default, Game Maker automatically cycles through a sprite's subimages to create an animation. However, this is not what is needed for this game—we need Game Maker to select the appropriate subimage based on the direction the rocket is moving in.

This requires a small amount of mathematics. There are 72 images representing a turn of 360 degrees, so each image must have been rotated by 5 degrees more than the last (because 360/72 = 5). Game Maker stores the direction of all objects in degrees, so it can work out which rocket subimage to use by dividing the rocket object's current direction by 5. Therefore we can make the rocket face in the right direction by using this rule (direction/5) to set the current subimage in a **Change Sprite** action.

Including a change sprite action in the landed object:

1. With the landed rocket Object Properties form open, include a **Change Sprite** action (**main1** tab) in its **End Step** event. Choose the landed rocket sprite from the menu and type `direction/5` into the **Subimage** property. `direction` is a special term that Game Maker recognizes as meaning the direction that this instance is currently facing in. Finally, set **Speed** to `0` to stop the sprite from animating on its own and changing the subimage. Figure 3-11 shows how this action should now look.

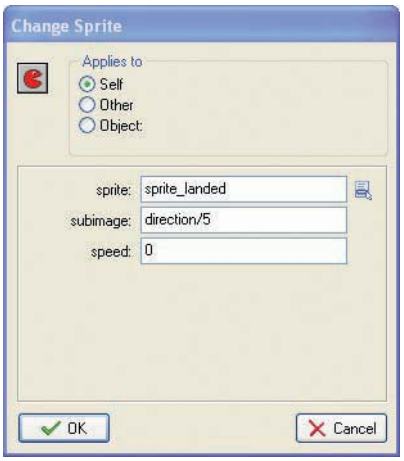

Figure 3-11. *Set the correct subimage in the sprite.*

■ **Note** This way of dealing with rotated images might seem rather clumsy, but many old arcade games were made in a similar way so that each rotated image could include realistic lighting effects. Nonetheless, the registered version of Game Maker contains an additional action to rotate a sprite automatically without the need for subimages at all.

We can also make use of this special term for the object's direction to add actions that allow the player to control the direction of the rocket using the arrow keys.

Including keyboard events for the landed rocket object:

1. Add a **Keyboard, <Left>** event to the landed rocket object.

2. Include the **Move Free** action and type `direction+10` in the **Direction** property. This indicates that the current direction should be increased by 10 degrees. Set **Speed** to `0` because we don't want the rocket to move independently of the special moon. This action should now look like Figure 3-12.

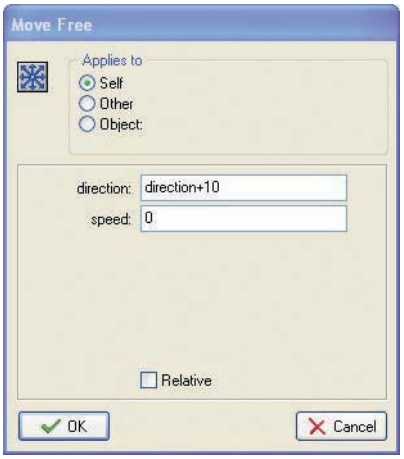

Figure 3-12. *Set the direction to equal itself plus 10.*

3. Add a similar **Keyboard** event for the **<Right>** key. Include a **Move Free** action and type `direction-10` in the **Direction** property.

The last control we need for the landed rocket will allow the player to launch the rocket using the spacebar. This control will need to turn the landed rocket object into a flying rocket object, but we can't make an action for this as we haven't created the flying rocket object yet! So we'll make the flying rocket now and come back to this later.

Creating the flying rocket object:

1. Create a new object called `object_flying` and select the flying rocket sprite. Set **Depth** to `-10` to make sure that this object appears in front of moons.

2. Add an **Other, Outside Room** event and include a **Wrap Screen** action to wrap around the screen in both directions.

3. Add an **End Step** event. Include a **Change Sprite** action, choose the flying rocket sprite, type `direction/5` in the **Subimage** property, and set the **Speed** to `0`.

4. Add a **Keyboard, <Left>** event and include a **Move Free** action. We don't want the player to have too much control over the flying rocket, so type `direction+2` in **Direction** and set **Speed** to `6`.

5. Add a **Keyboard**, **<Right>** event with a **Move Free** action. Type `direction-2` in **Direction** and set **Speed** to `6`.

The basic gameplay is nearly there now—just a few more events to tie up. First, the game should end when the rocket hits an asteroid. Next, when the flying rocket reaches a moon, it should turn into a landed rocket, and the moon should turn into the special moon (so that the landed rocket can follow it). We achieve this using the **Change Instance** action, which basically turns an instance from one type of object into another. To return to our jelly comparison, this is a bit like melting down the jelly from one instance and putting it into a new object mold. Although the instance may end up as a completely different kind of object, it keeps many of its original properties, such as its position on the screen and its direction. The fact that these values remain the same is critical—otherwise the launch direction of the landed rocket would get reset as soon as it turned into a flying rocket!

Adding collision events to the flying rocket object:

1. Add a **Collision** event with the asteroid object and include the **Restart Game** action (**main2** tab) in the **Actions** list. Later on we'll include an explosion to make this more interesting.

2. Add a **Collision** event with the moon object and include the **Change Instance** action (**main1** tab). Set the object to change into `object_landed` using the menu button, and leave the other options as they are.

3. Include a second **Change Instance** action for changing the moon into a special moon object. To make this action change the moon object (rather than the rocket), we need to switch the **Applies to** option from **Self** to **Other**. This makes the action apply to the other object involved in the collision, which in this case is the moon. Set the object to change into `object_specialmoon`. Figure 3-13 shows the settings.

Figure 3-13. *Change the other instance involved in the collision into a special moon.*

Finally, we can go back to the landed rocket object. This will need an event that changes it into a flying rocket and deletes the special moon when the spacebar is pressed.

Adding a key press event to the landed rocket object:

1. Reopen the Object Properties form for the landed rocket by double-clicking on it in the resource list.

2. Add a **Key Press, <Space>** event and include a **Move Free** action to set the rocket in motion. Type `direction` in the **Direction** property (this keeps the direction the same) and set **Speed** to `6`.

3. Now include a **Change Instance** action and change the object into an `object_flying`.

4. Finally, we want to delete the special moon because it no longer needs to be visited. Include a **Destroy** action and change the **Applies to** option to **Object**. Click the menu button next to this and select the `object_specialmoon`, as shown in Figure 3-14.

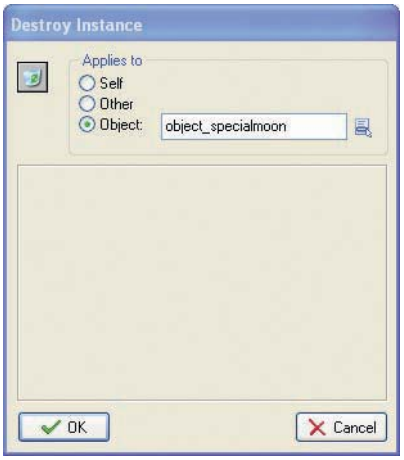

Figure 3-14. *Include a **Destroy** action for the special moon.*

Caution Using the **Object** setting for **Applies to** performs an action on all instances of that kind of object in the room. Deleting all of the special moon instances is fine in this case (as there is only one), but you will need to think carefully about the effects this setting will have before using it in your own games.

That completes the second version of our game. Make sure you save it and check that it all works as it should so far. You should now be able to rotate the rocket on a moon, launch it with the spacebar, and steer through the asteroids to land on another moon. Moons should disappear as you visit them, and the game should restart if you hit an asteroid. If something is not working, then check the instructions again, or compare your version with the version on the CD (`Games/Chapter03/galactic2.gm6`).

There are clearly a number of things still missing from the game, but the game is already quite fun to play. In the next section, we will add a scoring mechanism and a high-score table, as well as advancing the player to a new level once mail has been delivered to all the moons.

Winning and Losing

In this section we'll put a bit more effort into what happens when the player wins or loses the game. Let's begin by making asteroids a bit more explosive!

An Explosion

To get this working, we'll add a new explosion object and create an instance of it when the rocket hits an asteroid. This will play the explosion sound when it is created and end the game with a high-score table after the explosion animation has finished.

Adding an explosion object to the game:

1. Create a new object called `object_explosion`, and select the explosion sprite. Give it a **Depth** of `-10` to make it appear in front of other instances.

2. Add a **Create** event and include a **Play Sound** action (**main1** tab) for the explosion sound.

3. Add an **Other, Animation End** event. This event happens when a sprite reaches the final subimage in its animation.

4. Include the **Show Highscore** action (**score** tab) in the **Actions** list for this event. To make the high-score list look more interesting, set **Background** to the same as the background for the game, set **Other Color** to yellow, and choose a different font (e.g., Arial, bold). The action should now look like Figure 3-15.

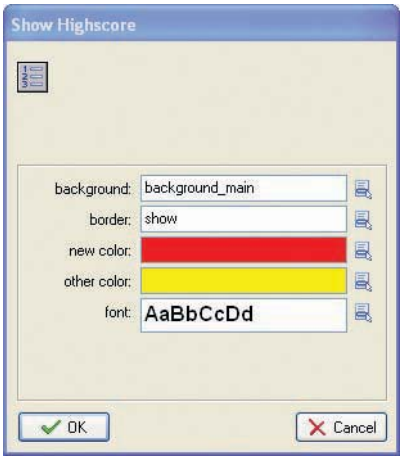

Figure 3-15. *You can spice up the high-score table.*

5. Also include a **Restart Game** action to start the game again after the high-score table is closed (**main2** tab).

6. Click **OK** to close the object.

Next we have to change the behavior of the flying rocket when it hits an asteroid.

Editing the flying rocket object:

1. Reopen the properties form for the flying rocket object by double-clicking on it in the resource list.

2. Select the **Collision** event with the asteroid by clicking on it once. Click once on the **Restart Game** action and press the Delete key to remove it from the action list.

3. Include a **Create Instance** action (**main1** tab) in its place, and set it to create the explosion object. Make sure the **Relative** property is enabled so that the explosion is created at the current position of the rocket.

4. Include a **Destroy Instance** action (**main1** tab) and leave it set to **Self** so that the rocket gets deleted. Click **OK** on the properties form to finish.

You might want to run the game now to see how it looks. Try colliding with an asteroid and you should get an explosion followed by the high-score table. Unfortunately, you can't score any points yet, so let's add this next.

Scores

If you've played the game quite a bit already, then you may have noticed a way of "cheating." You can avoid the risk of hitting asteroids by waiting for another moon to fly right next to your own and then quickly hop between moons. The game can become a lot less fun once this technique has been discovered, so our scoring system is designed to discourage the player from playing this way. Although they receive points for delivering mail, they also lose points for waiting on moons. This means that a player that takes risks by launching their rocket as soon as possible not only will have the most enjoyable playing experience but will also score the most points.

Editing game objects to include scoring:

1. Reopen the properties form for the special moon object and select the **Game Start** event. Include a **Set Score** action with a **New Score** of 1000. This gives the player some points to play with at the start. Close the properties form.

2. Reopen the properties form for the landed rocket and select the **End Step** event. Include a **Set Score** action with **New Score** as -1 and the **Relative** option enabled. This will repeatedly take 1 point off the score for as long as the player remains on a moon. As there are 30 steps every second, they will lose 30 points for every second of hanging around. Close the properties form.

3. Reopen the properties form for the flying rocket and select the **Collision** event with the moon object. Include a **Set Score** action with a **New Score** of `500` and the **Relative** option enabled.

4. Include a **Play Sound** action after setting the score and select the bonus sound.

Levels

At the moment, there is no reward for delivering all the mail. In fact, once all the moons are removed, the rocket just flies through space until it collides with an asteroid! This seems rather unfair, and it would be much better if the player advanced to a more difficult level. Making multiple levels in Game Maker is as simple as making new rooms. We can use actions to move between these rooms, and include more asteroids in the later levels to make them more difficult to play.

Let's begin by creating the new levels. You'll repeat these steps to make two more levels so that there are three in total. You can always add more of your own later on.

■ **Note** The order of the rooms in the resource list determines the order of your levels in the game, with the top level being first and the bottom level last. If you need to change the order, just drag and drop them into new positions into the list.

Creating more level resources for the game:

1. Right-click on a room in the resource list and choose **Duplicate** from the pop-up menu. This will create a copy of the level.
2. Go to the **settings** tab and give the room an appropriate name (`room_first`, `room_second`, etc.).
3. Switch to the **objects** tab, and add or remove instances using the left and right mouse buttons.
4. Make sure that each level contains exactly one special moon and one instance of the landed rocket.

In order to tell Game Maker when to move on to the next room, we have to be able to work out when there are no moons left in the current one. To do this, we will use a *conditional action* that asks the question "Is the total number of remaining moons equal to zero?" If the answer is yes (or in computer terms, **true**), then a block of actions will be performed; otherwise the answer is no (or **false**), and this block of actions is skipped. We'll put this check in the collision event between the flying rocket and the moon, so that players complete the level as soon as they hit the final moon.

■ **Note** All conditional actions ask questions like this, and their icons are octagon-shaped with a blue background so that you can easily recognize them.

Editing the flying rocket object to test for the number of remaining moons:

1. Reopen the properties form for the flying rocket and select the **Collision** event with the moon object.

2. At the end of the current list of actions, include the **Test Instance Count** action (**control** tab). Set the **Object** field to `object_moon` and the other settings will default to how we need them (**Number**, 0 and **Operation**, Equal to). This is now equivalent to the question "Is the total number of remaining moons equal to zero?" The form should look like Figure 3-16.

Figure 3-16. *We use the* ***Test Instance Count*** *action to count the number of moons.*

3. Below this action we need to start a *block*. A block indicates that a number of actions are grouped together as part of a conditional action. This means that all of the actions in the block will be performed if the condition is true and none of them if it is not. Add the **Start Block** action (**control** tab) directly below the condition to test the instances.

4. First, we will pause for a moment to give the player a chance to notice they have reached the final moon. Include the **Sleep** action (**main2** tab) and set **Milliseconds** to `1000`. There are 1,000 milliseconds in a second, so this will sleep for 1 second.

5. We'll award the player an extra bonus score of 1,000 points when they finish a level. Include a **Set Score** action (**score** tab) with a **New Score** of `1000` and make sure that the **Relative** option is enabled.

6. Include the **Next Room** action from the **main1** tab to move to the next room. No properties need to be set here.

7. Finally, add the **End Block** action (**control** tab) to end the block of the conditional action. The completed set of actions should now look like Figure 3-17. Note that the actions in the block are indented so that you can easily see that they belong together.

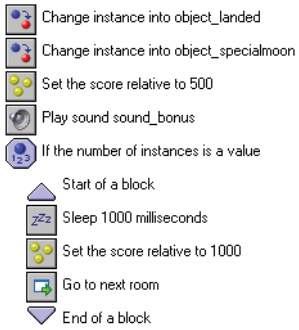

Figure 3-17. *Note that the actions in the block are indented.*

It is time to try out the game again. Save and play the game to check that you can go from one level to the next by visiting all the moons. You can also load this version of the game from the file `Games/Chapter03/galactic3.gm6` on the CD. However, if you complete the game you'll get an error message indicating that it has run out of levels. Don't worry—this is something we will fix in a moment, when we add some more finishing touches to the game.

Finishing Touches

To finish our game, we'll add an opening title screen, a help screen, and a congratulatory message upon completing the game. We'll also include a few visual touches to add a little bit of variety in the moons and asteroids.

A Title Screen

To create the title screen, we need a new object to display the name of the game and perform some initial tasks. We'll make it start the music and set the initial score, and then wait for the player to press a key before taking them to the first level.

Creating a new title object resource for the game:

1. Create a new sprite called `sprite_title` using `Title.gif`.
2. Create a new object called `object_title` and give it this sprite. Set the **Depth** property to `1` so that the moons go in front of it and the asteroids behind.
3. Add a **Create** event. This will contain the actions to start the music and set the score, but we've already created these in the special moon object, so we can simply move them over.

4. Open the special moon Object Properties form from the resource list and select the **Game Start** event to view its actions.
5. Drag and drop the two actions from the special moon **Game Start** event into the **Create** event of the title object. The **Game Start** event in the special moon should now be empty, and so it will delete itself automatically when the Object Properties form is closed. Do this now by clicking **OK** on the special moon's properties form.

6. Add a **Key Press, <Any key>** event to the title object and include the **Next Room** action in the action list for this event (**main1** tab).

Next we need to create a new room for the title screen.

Creating a new title room resource for the game:

1. Create a new room called `room_title` and give it an appropriate caption. Also set the room's background in the same way as before.
2. Add a few moon and asteroid instances to the room (just for effect).
3. Place an instance of the new title screen object in the center of the room.
4. Close the room properties.
5. To make sure that this is the first room in the game, drag the new room to the top of the list of rooms in the resource list.

Now quickly test the game to check that this all works correctly.

Winning the Game

We also need to stop the game from producing an error at the end and congratulate the player instead. Similar to how we created the title room, we will create a finish room with a finish object to display the message and restart the game.

Creating a new finish object resource for the game:

1. Create a new object called `object_finish`. It doesn't need a sprite.

2. Add a **Create** event to the object and include the **Display Message** action in it (**main2** tab). Set the **Message** to something like: "Congratulations! You've delivered all the mail."

3. Include a **Set Score** action, with a **New Score** of `2000` and the **Relative** option enabled.

4. Include the **Show Highscore** action, with **Background, Other Color**, and **Font** properties set as before.

5. Finally, include the **Restart Game** action.

Now that we have the object, we can create a room for it to go in.

Creating a new finish room resource for the game:

1. Create a new room and place one instance of the new finish object inside it. As this object has no sprite, it will appear as a blue ball with a red question mark on it. This will not appear in the game, but it reminds us that this (invisible) object is there when we are editing the room.

Now test the game to check that you can complete it—and that you get the appropriate message when you do (in other words, not an error message!)

Adding Some Visual Variety

At the moment all the moons look exactly the same, and the asteroids even rotate in unison as they move around the screen. However, with a different moon sprite and a little use of the random command, we can soon change this.

Editing the moon and asteroid objects:

1. Open the properties form for the moon object and click the **Edit** button below the name of the object's sprite (this is just another way of opening the moon sprite's properties).
2. In the moon sprite's properties, click **Load Sprite** and select `Bases.gif` instead of the existing sprite. This sprite contains eight subimages showing different kinds of inhabitations on each moon. Click **OK** to close the Sprite Properties form.
3. Back in the moon Object Properties form, select the **Create** event and include a new **Change Sprite** action. Select the moon sprite and type `random(8)` in the **Subimage** property. This will randomly choose one of the inhabited moon sprites. Also set **Speed** to `0` to stop the sprite from animating on its own and changing the subimage.
4. Close the Action Properties and the moon Object Properties forms.
5. Include an identical **Change Sprite** action to the **Create** event of the special moon object in the same way. There is no need to edit the moon sprite again, as both objects use the same one.
6. Open the properties form for the asteroid object and include a new **Change Sprite** action in its **Create** event as well. This time choose the asteroid sprite, and type `random(180)` in the **Subimage** property. There are 180 images in the rotating asteroid animation, so this will start each one at a different angle. Also type `random(4)` in the **Speed** property so that asteroids rotate at different speeds.

Help Information

Once you have finished making a game, it is easy to sit back and bask in your own creative genius, but there is one more important thing you must do before moving onto your next game. It may seem blindingly obvious to you how to play your masterpiece, but remember that it is rarely that obvious to a newcomer. If players get frustrated and stuck in the first few minutes because they can't figure out the controls, then they usually assume it is just a bad

game rather than giving it the chance it deserves. Therefore, you should always provide some help in your game to explain the controls and basic idea of the game. Fortunately, Game Maker makes this very easy through its **Game Information**.

Adding game information to the game:

1. Double-click on **Game Information** near the bottom of the resource list.
2. A text editor will open where you can type any text you like in different fonts and colors.
3. Typically you should enter the name of the game, the name of the author(s), a short description of the goals, and a list of the controls.
4. When you're done, click the green checkmark at the top left to close the editor.

That's all there is to it. When the player presses the F1 key during game play, the game is automatically paused until the help window is closed. Test the game one last time to check that this final version works correctly. You can also load the final version of the game from `Games/Chapter03/galactic4.gm6` on the CD.

Congratulations

Congratulations! You've now completed your second game with Game Maker. You might want to experiment with the game a bit further before continuing as there is much more you could do with it. To start with, you could make more levels with faster-moving asteroids or smaller moons to make it harder to land on them. We've included both larger planet sprites and smaller planetoids for you to experiment with, so see what you can come up with.

This chapter has introduced you to more features of Game Maker. In particular you've made use of events and actions to change sprites and objects. You've also used the **Depth** property of objects to control the order in which the instances appear on the screen. This chapter has also introduced *variables* for the first time, even though we haven't called them that yet. For example, the word `direction` is a variable indicating the current direction of an instance. We also used the variables `x` and `y` that indicate the position of an instance. There are many variables in Game Maker, and they are extremely useful. We will see plenty more of them in the chapters to follow.

In the next chapter, we'll continue to build on what you've learned so far by creating a crazy action game that requires quick thinking to avoid being squished. It's amazing what can go on in a deserted warehouse . . .

CHAPTER 4

Target the Player: It's Fun Being Squished

Our third game will be an action game that challenges players to make quick decisions under pressure—and if they're not fast enough then they'll get squished! We'll introduce some new techniques for putting character animation into the game, and show how a *controller object* can be used to help to manage the game.

Designing the Game: Lazarus

As usual we'll need a description of our game. We've named it *Lazarus*, after the biblical character who was resurrected from the dead, because the game once had to be recovered from an old floppy disk that had become corrupted! Always remember to make backups of your data!

Lazarus has been abducted by the Blob Mob, who are intent on bringing this harmless creature to a sticky end. They've imprisoned him at the Blobfather's (sorry) factory, where they are trying to squish him under a pile of heavy boxes. However, they've not accounted for Lazarus's quick thinking, as the boxes can be used to build a stairway up to the power button that halts the machinery. Do you have the reactions needed to help Lazarus build a way up, or will the evil mob claim one more innocent victim?

Each level traps Lazarus in a pit of boxes stacked up on either side of the screen to contain him within the level. The arrow keys will move Lazarus left and right, and he will automatically jump onto boxes that are in his way. However, he can only jump the height of a single box, and stacks two or more boxes high will block his path. New boxes will periodically appear directly above Lazarus's current position and fall vertically down from the top of the screen until they come to rest. This means that the player will be able to use Lazarus's position to control where boxes fall and build a stairway up to the power button.

There will be four different types of boxes, increasing in weight and strength: cardboard, wood, metal, and stone. Falling boxes will come to rest on boxes that are stronger than them, but will crush boxes that are lighter. The type of each box is chosen at random, but the next box will be shown in the bottom-left corner of the window just before it appears. There will be a number of increasingly difficult levels, with higher stairways to build, and boxes that fall faster. When Lazarus gets squished, the level will restart to give the player another try. See Figure 4-1 for an example of how a level will look.

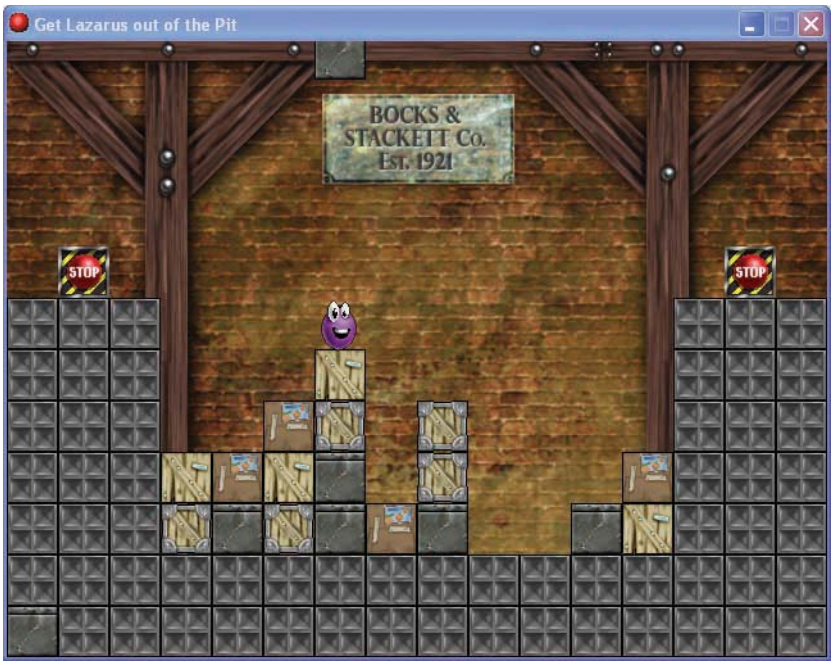

Figure 4-1. *This shows how a typical level might look in the Lazarus game.*

This may sound rather simple, but—as you'll see—it actually makes for a very challenging game! All the resources can be found in the `Resources/Chapter04` folder on the CD.

An Animated Character

Our first task will be to create the Lazarus character. We'll give him comical animations for when he's moving, jumping, and being squished to add to the appeal of the game. This will require a number of different sprites and several different Lazarus objects as well. Like the different behaviors of the rocket in Galactic Mail, using several objects helps us to separate the different animations of Lazarus in a simple way.

The animations for Lazarus have been designed around the size of the boxes in the game. All the boxes are exactly 40×40 pixels in size, so the animations that show Lazarus jumping from one box to the next need to be as tall and wide as two boxes (80×80 pixels). This means we'll be working with sprites of different sizes, and have to think carefully about where to place the origin of each sprite so that they match up correctly. Remember that Game Maker acts as if it is holding each sprite by its origin as it moves around the screen; so all the origins need to be at the same position relative to Lazarus—regardless of the size of the sprite. This should begin to make sense as you complete the steps that follow.

■ **Tip** Setting the **Smooth Edges** property in the Sprite Properties form can often make sprites appear less pixilated (blocky) in the game.

Creating the Lazarus sprite resources for the game:

1. Create a new sprite called `spr_laz_stand` using `Lazarus_stand.gif` from the `Resources/Chapter04` folder on the CD. This sprite is 40×40 pixels and shows Lazarus in his "normal" position. Remember that the origin for sprites defaults to the top-left corner (**X** and **Y** both set to `0`). We'll leave this where it is and make sure that the origins of all the other sprites match up with this position. Click the **OK** button to close the form.

2. Create another sprite called `spr_laz_right` using `Lazarus_right.gif`. This sprite is 80×80 pixels and shows Lazarus jumping 40 pixels to the right (use the blue arrows to preview the animation). As usual, the origin has defaulted to the top-left corner of the sprite, but the top-left corner is further above Lazarus's head than in the last sprite. To match up with the previous sprite, we need to move the origin down by 40 pixels—so set the **Y** value to `40`. The properties form should now look like Figure 4-2.

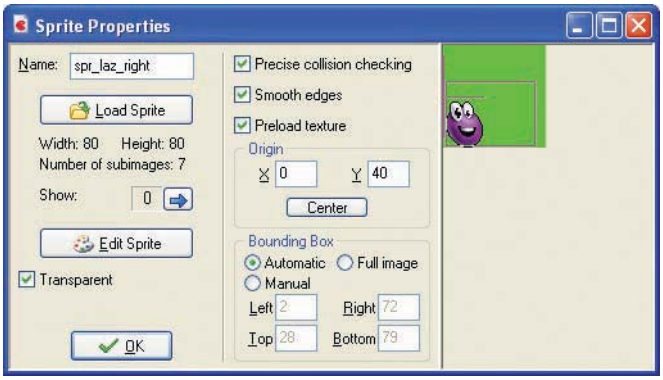

Figure 4-2. *The origin for this sprite has been moved to halfway down the left-hand side.*

3. Create a `spr_laz_jump_right` sprite in exactly the same way using `Lazarus_jump_right.gif` (with an **X** value of `0` and a **Y** value of `40`).

4. Create a `spr_laz_left` sprite using `Lazarus_left.gif`. This sprite is also 80×80 pixels and shows Lazarus jumping 40 pixels to the left, but this time Lazarus starts on the bottom-right side of the sprite. This means we need to move the origin 40 pixels down and 40 pixels right to place it at the same relative position as before. Set both the **X** and **Y** values to `40` and close the form.

5. Create a `spr_laz_jump_left` sprite in exactly the same way using `Lazarus_jump_left.gif` (with an **X** value of `40` and a **Y** value of `40`).

6. Create two more sprites called `spr_laz_afraid` and `spr_laz_squished` using `Lazarus_afraid.gif` and `Lazarus_squished.gif`. These are 40×40 pixels so there's no need to change the origin.

These are all the sprites we need for Lazarus, so our next step is to make some objects for him. The main object will be the "normal" standing Lazarus. This is the most important one, as it will react to the player's keyboard input. The others are only there to play the different

animations, after which they turn themselves back into the standing Lazarus. They will also move Lazarus to a new position that corresponds to the final frame of the animation.

We have a bit of a chicken-and-egg situation here, as we did with the two types of rockets in Galactic Mail. The "normal object" will need actions to turn it into "animating objects" (which don't exist yet) and the "animating objects" will need actions to turn them into "normal objects" (which also don't exist yet). So which objects do we create first? Well, the answer is that we create the "normal object" but come back to creating its events and actions after we have created the "animating objects"—crafty, eh?

Creating Lazarus object resources for the game:

1. Create a new object called `obj_laz_stand` and give it the standing Lazarus sprite.
2. Press **OK** to close the properties form (we will come back to it later).
3. Create a new object called `obj_laz_right` and give it the sprite that hops one box horizontally to the right (`spr_laz_right`).
4. Add an **Other, Animation End** event. Remember that this event happens when a sprite reaches the last subimage in its animation.

5. Include the **Jump to Position** action in this event (**move** tab). Set **X** to `40` and **Y** to `0`, and make sure that the **Relative** option is enabled. As the boxes are all 40×40 pixels, this will move Lazarus exactly one box to the right at the end of the animation.

6. Also include the **Change Instance** action (**main1** tab) below this and select `obj_laz_stand` as the object to change back into.
7. Click **OK** to close the object properties form.

8. Create another object called `obj_laz_left` and give it the sprite that hops one box horizontally to the left (`spr_laz_left`). Repeat the same process as before (steps 4–7), but set **X** to `-40` for the **Jump to Position** action.

9. Create another object called `obj_laz_jump_right` and give it the sprite that hops up one box diagonally to the right (`spr_laz_jump_right`). Repeat the process, setting **X** to `40` and **Y** to `-40`.

10. Add a final object called `obj_laz_jump_left` and give it the sprite that hops up one box diagonally to the left (`spr_laz_jump_left`). This time, set **X** to `-40` and **Y** to `-40`.

We may as well get the squished Lazarus object out of the way now too—even though we won't need it for a while. Once its gruesome animation finishes, this object will display a message to tell the player that they've been squished. This isn't because we think they are too stupid to notice, but it provides a useful pause before starting the level again! We're not going to add lives or high scores in this game, so we'll simply restart the level to give the player another try.

Creating the squished Lazarus object resource:

1. Create an object called `obj_laz_squished` and give it the squished Lazarus sprite.

2. Add an **Other, Animation End** event and include the **Display Message** action (**main2** tab) in it.

3. Type something like "`YOU'RE HISTORY!#Better luck next time`" into the message properties. Note that putting the # symbol in the middle of the message will start a new line from that point.

4. Finally, include the **Restart Room** action (**main1** tab) after the message action and press **OK** to close the object properties form.

Okay, so now we have these animation objects in place, we can continue making the main standing Lazarus object we started on the previous page. One of its main jobs is to change into the appropriate animating object when the player presses a key. The appropriate object depends on whether Lazarus is standing next to any boxes. We'll use a conditional collision action to help Game Maker work this out for us.

Adding a right key event for the standing Lazarus object:

1. Reopen the properties form for the `obj_laz_stand` object by double-clicking it in the resource list.

2. We'll start by creating actions to handle moving to the right. Add a **Key Press, <Right>** event and include the **Check Collision** action in it (**control** tab).

3. This action allows us to check that there *would* be a collision if we moved this instance to a particular position on the screen. We need to make sure that Lazarus is on solid ground before allowing him to move, as he shouldn't be able to jump when he is standing on thin air! To check that this is the case, we set **X** to `0` and **Y** to `8` (slightly below his current position), and enable the **Relative** option.

Note Conditional collision actions have an **Objects** option, which allows us to choose between checking for collisions with all objects or only ones marked as solid. We're leaving this set to only solid, so we need to remember to set the **Solid** property later when we create the box objects.

4. All of the remaining actions in this event depend on the previous condition (they only need to be called if it is **true**). Consequently, we'll need to include them all between **Start Block** and **End Block** actions. Include the **Start Block** action now.

5. Now include the **Check Empty** conditional action. This conditional action is the opposite of the last one: it checks that there *wouldn't* be a collision if we moved to a particular position on the screen. So to check that the space to the right of Lazarus is free, set **X** to `40` (the width of a box), set **Y** to `0`, and enable the **Relative** option.

6. Include the **Change Instance** action (**main1** tab) and select the `obj_laz_right` object. Select **yes** to **Perform Events**. This means that the **Create** event of the object we're turning into *will* get called (which is important later when we add sound effects).

■ **Note** The **Perform Events** option controls whether the **Destroy** event of the current object and the **Create** event of the new object should be called. This isn't usually necessary so it does not call them by default.

7. Next include the **Else** action from the **control** tab (more about this in a moment).

8. Include another **Check Empty** conditional action directly after this. This should verify that there are no boxes diagonally, up, and to the right of Lazarus. Set **X** to `40` and **Y** to `-40`, and enable the **Relative** option.

9. Next include the **Change Instance** action and select the `obj_laz_jump_right` object. Select **yes** to **Perform Events**.

10. Finally, include an **End Block** action to conclude the actions that should be performed if Lazarus is on solid ground. The list of actions now should look like Figure 4-3.

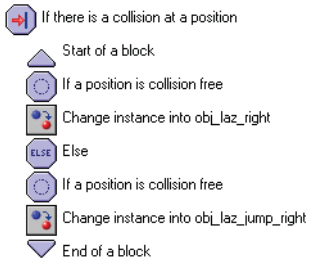

Figure 4-3. *Here are the actions for moving or jumping to the right.*

This is the first time we've used the **Else** action, but it is often used alongside conditional actions in this way. On its own, a conditional action only allows you to specify actions that should be performed if a condition is true. However, in combination with **Else**, you can specify different actions to be performed if that same condition is not true. This has many uses, but in this situation it allows us to ask sequences of questions like this:

> *Is there solid ground beneath Lazarus's feet? Yes. Well, is there a free space to the right of Lazarus? No—there's a box in the way. Okay, well, is there a free space on top of that box then? Yes—let's jump on top of it.*

This is just one possible outcome, but our actions provide outcomes for four different situations: not moving when falling through the air; moving horizontally to the right when no boxes are in the way; jumping diagonally to the right when a single box is in the way; and doing nothing at all when more then one box is in the way. You can think of this action list as reading something like this:

If the position below has something solid in it, then read the next sentence. If the position to the right is collision free, then change into object `obj_laz_right`*; else, if the position diagonally right is collision free, then change into object* `obj_laz_jump_right`.

Before continuing, go through the actions step by step in your head and try to work out how you end up with each of these different outcomes (move right, move diagonally right, and no movement). When you're happy that this makes sense, we'll move on and do the same thing for the left arrow key.

■ **Note** Like other conditional actions, the **Else** action can be used with or without blocks. If blocks are not used, then the **Else** only affects the action that immediately follows it.

Adding a left key press event to the standing Lazarus object:

1. Add a **Key Press, <Left>** event and include the **Check Collision** action. Set **X** to `0` and **Y** to `8`, and enable the **Relative** option (this checks below).

2. Include a **Start Block** action.

3. Include the **Check Empty** conditional action (**control** tab) with **X** set to `-40`, **Y** set to `0`, and the **Relative** option enabled (this checks left).

4. Next, include a **Change Instance** action (**main1** tab) and select `obj_laz_left`. Choose **yes** to **Perform Events**.

5. Now include **Else** action from the **control** tab.

6. Include a **Check Empty** action with **X** set to `-40`, **Y** set to `-40`, and **Relative** enabled (this checks diagonally left).

7. Include a **Change Instance** action and select the `obj_laz_jump_left` object. Choose **yes** to **Perform Events**.

8. Finally, include an **End Block** action to finish the block of actions.

Although our keyboard events stop Lazarus from jumping in mid-air, there aren't yet any events to make him fall down to the ground when he is. We'll get Game Maker to test for this in a **Step** event so that it is continually checking to see if he should be falling. However, we need to think carefully about how far he should fall in each step. The amount of movement in each step will determine how fast he falls, but it will make our job much simpler if we also choose a number that divides exactly into 40 (the height of the boxes). Can you think why?

Let's imagine that we chose a number that doesn't divide into 40, like 12. Lazarus would have fallen 12 pixels after one step, 24 pixels after two steps, 36 pixels after three steps, and 48 pixels after four. At no stage has Lazarus fallen the exact 40 pixels needed to fall the height of one box; he is either 4 pixels too high (at 36 pixels) or 8 pixels too low (at 48 pixels). This means he would either end up floating above boxes, or jammed someway into them! Using any number that divides into 40 will avoid this problem (1, 2, 4, 5, 8, 10, 20, or 40), so we've chosen a value of 8 because it produces a sensible-looking falling speed.

Adding a step event to the standing Lazarus object to make it fall:

1. Add the **Step, Step** event to the standing Lazarus object. We are using the "standard" **Step** event as we don't really care exactly when Lazarus falls, provided he does.

2. Include a **Check Empty** action in the **Step** event**,** setting **X** to `0` and **Y** to `8`, and enabling the **Relative** option. This action checks for empty space just below Lazarus.

3. Include a **Jump to Position** action directly after it so that it will only be performed if the **Check Empty** condition is true. We need to give it the same relative settings as before, so that it moves into the empty space. Set **X** to `0` and **Y** to `8`, and enable the **Relative** option.

A Test Environment

We've gone through quite a lot of steps so far without being able to test our work, so before going any further let's quickly create a test level for Lazarus to move around in. There are no falling boxes yet, so we'll have to create some random stacks of our own to check if the movement is working correctly. We'll create just one box type to do this: the boxes that make up the walls of the pit.

Creating the wall object resource for the game:

1. Create a new sprite called `spr_wall` using `Wall.gif`. Disable the **Transparent** option as the walls for this level need to look completely solid.
2. Create a new object called `obj_wall` and give it the wall sprite. Enable the **Solid** option so that the checks in the standing Lazarus object can detect the wall.
3. Create a new room called `room_test` and provide a caption in the **settings** tab.
4. Look in the toolbar at the top of the Room Properties form and set both **Snap X** and **Snap Y** to `40`. All our boxes are 40×40 pixels, so this will help us to place them neatly on the level. The grid in the room will change accordingly.
5. Switch to the **objects** tab again and select the wall object to place. Create a level with a number of boxes that form flat areas and staircases (remember, you can hold the Shift key to add multiple instances). Also add one instance of the standing Lazarus object. Try to make it look something like Figure 4-4.

■ **Note** Sometimes when you close a room form you get a warning message saying that there are instances outside the room. This can happen when you accidentally move the mouse outside the room area while adding objects. You will be asked whether these instances should be removed—simply click the **Yes** button.

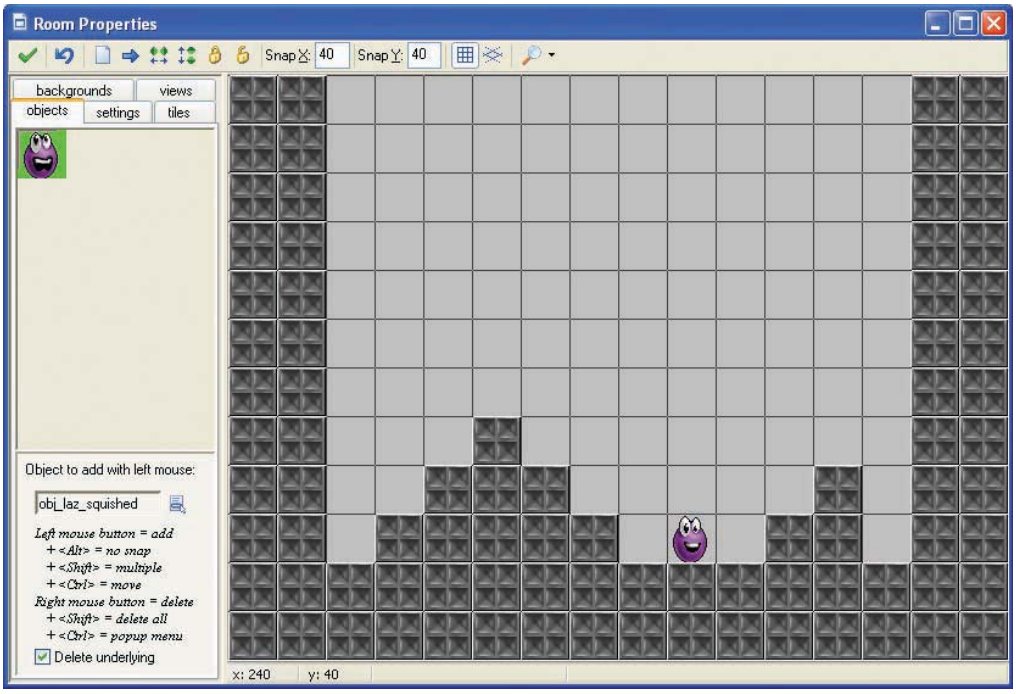

Figure 4-4. *Your test level should look something like this.*

At last, you can finally run the game! Test the character's movement in all the possible situations and make sure that he behaves the way you would expect. If something isn't working right, then check your steps carefully, making sure that you enabled the **Relative** option in all the actions where it was indicated. Alternatively, you can load the current version from the file `Games/Chapter04/lazarus1.gm6` on the CD.

Falling Boxes

Our next goal is to create the falling boxes that both threaten the player and provide the means for their escape. As indicated in the game description, there will be four types of boxes in the game: stone boxes, metal boxes, wooden boxes, and cardboard boxes. As you would expect, stone boxes are the heaviest and cardboard boxes are the lightest. Falling boxes are chosen at random and heavier boxes will crush lighter boxes as they fall—making it harder to build a stairway out of the pit. However, to give the player a chance to think ahead, the next box will be shown in the corner of the screen while the last box is still falling.

Each box will need to change its behavior three times in the game: first it appears in the corner, as the "next box"; then it falls down the screen until it lands on another box; and finally it forms a stationary obstacle for Lazarus to negotiate. As you may have guessed, we will achieve this by creating three different objects for each box: one for each behavior. We will start by creating the stationary boxes, as they are the simplest to make. First, though, we need to create some new sprites.

Creating new box sprite and object resources for the game:

1. Create sprites called `spr_box_stone` and `spr_box_card` using `StoneBox.gif` and `CardBox.gif`. Disable the **Transparent** option on both these sprites.
2. Now create sprites called `spr_box_metal` and `spr_box_wood` using `MetalBox.gif` and `WoodBox.gif`. This time leave the **Transparent** option enabled, as these two sprites have a small amount of transparency around the edges.
3. Create a new object called `obj_box_stone` and give it the sprite for the stone box. Set the **Solid** option so that it is detected in collision tests.
4. Repeat the previous step to add objects for `obj_box_metal`, `obj_box_wood` and `obj_box_card`.

Next we'll make the falling boxes. These need to start at the top of the screen, directly above Lazarus's horizontal position, so we'll make use of the `x` variable of the standing Lazarus object to tell us where that is. Once it starts falling, we'll give it a speed of 5 because that divides exactly into 40 (important for the same reasons as before) and it is slightly slower than the speed that Lazarus falls (otherwise a box might squish Lazarus in the air!). When a box collides with a heavier box, it turns into a stationary box, but when it collides with a lighter box, it destroys that box and continues to fall.

Creating falling box objects for the game:

1. Create a new object called `obj_falling_stone`, give it the sprite for the stone box, and select the **Solid** option as before.

2. Add a **Create** event and include a **Jump to Position** action in it. Type the variable `obj_laz_stand.x` (the horizontal position of Lazarus) into **X** and set **Y** to `-40`. This will make the box start above Lazarus, just out of view at the top of the screen.

3. Next include the **Move Fixed** action, using a downward direction and a **Speed** of `5`.

4. Add a **Collision** event with `obj_laz_stand` and include a **Change Instance** action in it. Change the **Applies to** option to **Other**, so that it changes Lazarus rather than the box. Select the `obj_laz_squished` and select **yes** to **Perform Events.**

5. Add another **Collision** event, this time with `obj_wall`. This needs to stop the box moving, so include a **Move Fixed** action and select the middle square with a **Speed** of `0`. Also include a **Change Instance** action, and select the stationary box `obj_box_stone`.

6. Add a third **Collision** event with `obj_box_stone` and include the same two actions as the **Collision** event with the wall above (you could copy them).

7. Add a fourth **Collision** event with `obj_box_metal`. The metal box is lighter than the stone box so it must be crushed. Include a **Destroy Instance** action and select the **Other** object.

8. Add fifth and sixth **Collision** events with `obj_box_wood` and `obj_box_card`, both including identical **Destroy Instance** actions as we did in step 7 to destroy the **Other** box in the collision.

Okay, that's one of the falling boxes. The other falling boxes are similar but need to behave slightly differently when they collide with different kinds of boxes.

9. Create the remaining three falling objects for the other types (`obj_falling_metal`, `obj_falling_wood`, and `obj_falling_card`). Repeat steps 1–8 for each one, using step 7 when a box crushes another box and step 5 when a box stops moving. Refer to Table 4-1 when deciding which boxes should crush each other.

Table 4-1. *Box Materials That Should Crush Each Other*

Material	Material(s) That It Crushes
Stone	Metal, Wood, and Card
Metal	Wood and Card
Wood	Card
Card	None

Phew! That was quite a lot of work (28 events and 46 actions), made worse by the fact that we had to repeat the same steps over and over again. In Chapter 6 we will see that there is actually a quicker way to do this kind of thing using *parents*. Nonetheless, although this might have seemed like a lot of effort, it may help you to appreciate the work that goes into a commercial game. They usually take at least 18 months to program and require hundreds of thousands of lines of code to make them work!

Now let's set about creating the final set of boxes that appears in the bottom-left corner to show the player which box is coming next. This adds an important element of gameplay, allowing the player to plan ahead and adapt their strategy based on where it would be most useful for the next box to fall. It requires quick thinking and takes a bit of practice, but it helps to create a challenging and rewarding game. The "next box" objects are very simple to make, but we'll need four of them again—one for each type of box.

Creating next box object resources for the game:

1. Create a new object called `obj_next_stone`, give it a stone box sprite, and enable the **Solid** option. That's it, so click **OK** to close the object properties.

2. Create objects for `obj_next_metal`, `obj_next_wood`, and `obj_next_card` in the same way.

I'm sure you'll be relieved to find out that's all the boxes we need to create for this game! However, while the falling boxes have actions to turn them into stationary boxes, there are no actions yet for turning next boxes into falling boxes, or creating next boxes in the first place. That's because we are going to create a *controller object* to do this. A controller object is usually an invisible object (it doesn't have a sprite), which performs important actions on other objects. Our controller object will use a **Step** event to continually check if there is a falling box on the level. If not, then it will turn the current next box into a falling box and create a new next box. In this way, the controller object will maintain a constant cycle of new and falling boxes until the level is completed—or the player gets squished!

Creating a controller object resource for the game:

1. Create a new object called `obj_controller` and leave it without a sprite.

2. Add a **Step, Step** event and include the **Test Instance Count** conditional action (**control** tab). This counts the number of instances of a particular object on the level and tests it against a value. Choose the `obj_falling_stone` object; leave **Number** as **0** and **Operation** as **Equal to**. This creates a condition that is true if the number of falling stone box instances on the level is equal to 0 (i.e., there aren't any!).

3. Include three more **Test Instance Count** conditional actions to check if there are no instances of `obj_falling_metal`, `obj_falling_wood`, and `obj_falling_card` in the same way. When combined, these conditional actions will make sure that there are no falling boxes of any kind on the level before creating a new one.

4. Include a final **Test Instance Count** action for the `obj_laz_stand` object, but set **Number** to `1` and the **Operation** to **Equal to**. This makes sure that there is an instance of the standing Lazarus object on the level, rather than any of the animating objects.

5. Include a **Start Block** action. This will group together all the actions that need to be performed to create the new box.

6. Include a **Change Instance** action and select **Object** for **Applies to,** so that it changes *all* instances of one kind of object on the level into another. Set **Object** to `obj_next_stone`, **Change Into** to `obj_falling_stone`, and select **yes** to perform events (the **Create** event for the falling box needs to be performed to start it in the correct position). This will turn any stone next boxes into stone falling boxes. The action should now look like Figure 4-5.

Figure 4-5. *Change the next box into a falling box.*

7. However, because the type of box will be chosen randomly, we don't know what kind of next box object the next box will actually be. To cover all bases, add three more **Change Instance** actions to change `obj_next_metal` objects into `obj_falling_metal` objects, `obj_next_wood` into `obj_falling_wood`, and `obj_next_card` into `obj_falling_card` in the same way.

8. Next we need to randomly create one of the next box objects. Include a **Create Random** action (**main1** tab) and select the four different next box objects. Set **X** to `0` and **Y** to `440`, and leave **Relative** disabled. Remember that when **Relative** is disabled, **X** and **Y** are measured from the top-left corner of the screen. These coordinates will therefore put the new next box where it should be in the bottom-left corner of the screen. The action should now look like Figure 4-6.

9. Finally, include an **End Block** action to conclude the block of actions that are dependent on all the conditions above them being true.

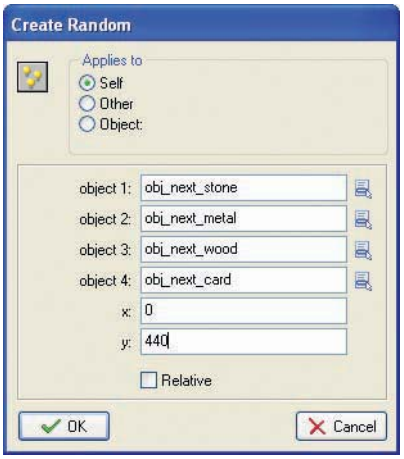

Figure 4-6. *The **Create Random** action allows us to randomly create one of the four next box objects.*

This long list of conditional actions means that the block of actions will only be performed if all these conditions are true. In other words, if there are no instances of `obj_falling_stone` and no instances of `obj_falling_metal` and no instances of `obj_falling_wood` and no instances of `obj_falling_card` and one instance of `obj_laz_stand`, *then* Game Maker will create a new box.

You might have thought it a bit odd that we need to check that there is a standing Lazarus object as part of our conditions for creating new boxes. If you look back at one of the **Jump To Position** actions in the **Create** events of the falling boxes, you will remember that we use the `obj_laz_stand.x` variable to start the object in the correct position. However, Game Maker can't provide that object's x position if it has turned into an animation object, so it will create an error in the program. So to avoid this possibility we check that there's a standing Lazarus instance on the level before creating new falling boxes.

Now it's finally time to test our new objects.

Editing the test room to add new instances:

1. Reopen the test room we created by double-clicking on it in the resource list.
2. Remove all the extra wall instances so that it leaves just a pit with walls on both sides and across the bottom.
3. Add one instance of the controller object into the room (easily forgotten!)

■ **Note** When an object has no sprite, it shows up in the Room Properties form as a blue ball with a red question mark. This will not appear in the game, but reminds us that this (invisible) object is there when we are editing the room.

Now run the game and test it carefully. Make sure that the box that appears in the bottom left is actually the box that falls down the screen next and check that heavier boxes are crushing lighter ones. As usual, if there are any problems, then carefully check the instructions or load the game from `Games/Chapter04/lazarus2.gm6` on the CD.

Finishing Touches

We now have all the basic ingredients of the game in place and there are just a few more things to do before we could call it a finished game. There's no way to complete a level yet, so we need to include the stop buttons that will halt the boxes and move the player onto the next level. Some sound effects would also be nice—as would a background and a title screen. We're obviously going to need a few different levels, too. However, before all that we're going to add something cool that will endear the player to Lazarus's plight a little more.

No Way Out!

You may have noticed that there's another animation we haven't used yet that shows Lazarus looking afraid. We're going to show this animation when he's in a hopeless situation and knows he is about to meet his end. However, rather than create a new object for this animation like we did with the others, we're just going to change the sprite of the standing Lazarus object when he becomes afraid. We can do this because "being afraid" does not need any actions of its own: it has exactly the same behavior as standing—it just looks different. We're going to control this animation within the **Step** event, so that the correct animation is chosen at any point in time. We will use **Check Collision** actions to detect if Lazarus is surrounded by stacks of boxes two or more high on both sides. The **Check Collision** action performs actions only when there *is* a collision at a particular point. In this way, we can detect whether Lazarus is trapped on all sides and set his animation to be afraid.

Editing the standing Lazarus object to detect for being trapped:

1. Reopen the standing Lazarus object and select its **Step** event, so that you can see the existing actions for this event.

2. Include the **Check Collision** conditional action (**control** tab) below the last action in the list. Set **X** to 40 and **Y** to 0, and enable the **Relative** option. This checks for a box to the right of Lazarus.

3. Include another **Check Collision** action with **X** set to 40, **Y** set to -40, and the **Relative** option enabled. This checks for a box diagonally to the right of Lazarus.

4. Include two more **Check Collision** actions: one with **X** set to -40 and **Y** set to 0, and the other with **X** set to -40 and **Y** set to -40. Both should have the **Relative** option enabled. These check for boxes to the left and diagonally to the left of Lazarus.

5. Finally, include a **Change Sprite** action, using the "afraid Lazarus" sprite. This will now only happen if the four conditional actions above are true and Lazarus is literally boxed in.

Hopeless as this situation may sound, it is actually possible for Lazarus to be saved from this predicament by a heavy block crushing the stack of boxes on one side of him. If this happens, then we would like Lazarus to stop being afraid. We *could* include conditional actions to check for this happening and change his sprite back to normal. However, we can achieve the same effect simply by including a **Change Sprite** action at the very beginning of the list of actions for this event. Changing into the standing Lazarus sprite by default will make the sprite revert back to normal if he stops being trapped.

Editing the standing Lazarus object to detect for being freed:

1. Select the **Step** event for the standing Lazarus object so that you can see the existing actions for this event.

2. Include a **Change Sprite** event at the very beginning of the list of actions (you can drag actions about if it falls in the wrong place). Set it to change into the standing Lazarus sprite.

You might want to play the game now and make sure that this new feature is working correctly. Features like this don't change the gameplay directly, but add to the playing experience and make the game more entertaining to play.

Adding a Goal

The player's goal is to reach one of the stop buttons, so that it halts the machinery and stops dropping the boxes. However, in practice all the buttons really need to do is move the player onto the next level when the standing Lazarus object collides with them. If there are no more rooms, then it will show a completion message and restart the game.

Creating a new button object resource for the game:

1. Create a new sprite called `spr_button` using `Button.gif`.
2. Create a new object called `obj_button` and give it the button sprite. Set **Depth** to `10` so that it appears behind other objects.

3. Add a **Collision** event with the standing Lazarus object and include a **Sleep** action in it (**main2** tab). Set **Milliseconds** to `1000` (1 second) and **Redraw** to true. This should give a brief pause for the player to realize they have completed the level.

4. Include a conditional **Check Next** action (**main1** tab).

5. Include a **Next Room** action (**main1** tab).

6. Include an **Else** action followed by a **Start Block** action.

7. Include a **Display Message** action (**main2** tab) and set **Message** to something like "`CONGRATULATIONS#You have completed the game!`"

8. Include a **Different Room** action and set **New Room** to the first room (which is the only room at the moment).

9. Finally, include an **End Block** action and close the object properties.
10. Edit your test room and add a stop button on either side at the top of the pit.

Starting a Level

At the moment, boxes start falling as soon as the player enters the level, leaving them with no time to gather their thoughts and prepare their strategy. We're going to help them out by creating a starter object that displays the title for a couple of seconds before changing itself into the controller object and starting to drop boxes.

Creating a new starter object resource for the game:

1. Create a new sprite called `spr_title` using `Title.gif`.
2. Create a new object called `obj_starter` and give it the title sprite.

3. Add a **Create** event and include a **Sleep** action in it. Set **Milliseconds** to `2000`, for a wait of two seconds.

4. Include the **Change Instance** action and select the controller object. Close the object properties.
5. Edit your test room, and remove the controller object using the right mouse button. Add the starter object at an appropriate place instead.

Note You may have noticed that the title doesn't appear on the first level when you run the game. This is because the starter object's **Create** event is executed before the window appears, so it has already turned into a controller object by the time we see the room. This can be remedied using an **Alarm** action to add a delay, but we won't worry about this for now, and we'll come back to alarms in Chapter 6.

Sounds, Backgrounds, and Help

It's about time we made the game feel a bit more professional by including sound effects and music in the game. This is quite simple and you can probably handle most of this on your own by now, but here are some pointers to help you on your way:

1. All the sound resources can be found in the `Resources/Chapter04` folder on the CD.

2. You'll need to add sounds for `Music.mp3`, `Wall.wav`, `Crush.wav`, `Squished.wav`, `Move.wav`, and `Button.wav` and play them at the right times using the **Play Sound** action (**main1** tab).

3. A good place to start playing the music would be in a new **Game Start** event for the controller object. You'll find the **Game start** event in **Other** events. Don't forget to set **Loop** to true in the **Play Sound** action to make the music loop forever.

4. You'll need to add crush or wall sound effects to the existing **Collision** events between falling box objects and stationary box objects.

5. Add a new **Create** event for the squished Lazarus object, and play the squished sound effect there. This will save you the trouble of putting it in each of the four collision events between falling boxes and Lazarus.

6. Adding **Create** events to play the move sound effects would also be a good way of handling the four moving Lazarus objects.

7. Finally, you'll need to play the button sound effect in the **Collision** event between the button and Lazarus.

Test the game and make sure all the sound effects are playing in the correct place. If you don't hear a sound when moving around, check that you set **Perform Events** to **yes** in the **Change Instance** actions that change into the animating objects. If you didn't, then Game Maker won't perform the **Create** events that contain the sound effects.

A backdrop to the levels would also improve the look of the game, and we should put together some kind of help text for the player too.

Creating a background resource and Game Information:

1. Create a background using `Background.bmp` from `Resources/Chapter04` on the CD.
2. Reopen the properties form for the room and select the **backgrounds** tab. Select the new background from the menu halfway down on the left.
3. Double-click on **Game Information** in the resource list and add a help text for the game. Remember to include the name of the game and who it was created by (you), along with a short description of the aims and controls.

Levels

All that is left now is to create a variety of levels for your game. We talk about level design in much more detail in Chapter 8, but it's probably best to start with shallow pits and buttons on each side to keep things fairly easy. However, as the levels progress they can become as deep and narrow as you like! Making the floor of the pit higher will make the level harder, as the player has less time to react to the falling boxes. You could also place stationary boxes in unhelpful places or place the buttons in mid-air to vary the challenge. One sure way to make the game more challenging is to increase the **Speed** setting on the **settings** tab for each level. This controls the number of steps per second on each level. It defaults to 30 steps per second, but higher numbers will make the game faster and harder and lower numbers will make it slower and easier.

Now it's up to you to create some interesting levels for the game. Remember that duplicating rooms will save you a lot of work, so right-click on the room in the resource list and select **Duplicate** from the pop-up menu. Once you've made your levels, let someone else try to play them and see how difficult they find it. Game designers often find their games very easy because they have played them so much, but it is often much harder for everyone else. This is something you should always try to bear in mind when designing your games.

One very last thing: you may find it helpful to add a cheat in your game that allows you to skip between levels. You can do this as follows.

Editing the controller object to add cheats:

1. Open the properties form for the controller object.

2. Add a **Key Press, <N>** event and include the **Next Room** action.

3. Add a **Key Press, <P>** event and include the **Previous Room** action.

Good luck, and don't forget to remove these cheats when the game is finally finished!

Congratulations

You'll find the final version of the game in the file `Games/Chapter04/lazarus3.gm6` on the CD. You might want to extend the game a bit further by adding opening and closing screens, or adding a scoring system to the game so that players can compete for the highest score. If you're feeling particularly adventurous, why not try adding some bonuses that sometimes appear when boxes are crushed by each other? One of these could even transform all the stationary boxes into stone boxes—or card boxes if you're feeling mean!

By making this game, you have learned how to animate characters, both by creating different objects and by switching sprites. You have also seen how to use a controller object to manage the game, plus you've learned how to use **Else** actions to provide extra control over the outcome of conditional actions. In fact, you've learned a lot about Game Maker over these past few chapters, and it's about time we gave you a bit of a break. With this in mind, the next chapter is all about game design and you won't have to go near events and actions again until Chapter 6. In the meantime, we'll be thinking more carefully about the designs behind the games we've made so far, and we'll be exploring what makes them fun to play.

CHAPTER 5

Game Design: Interactive Challenges

Once you've caught the game-making bug, then it's only a matter of time before you'll want to start designing your own games. There's nothing more satisfying than realizing your creative ideas and seeing other people enjoy them, and that's precisely what making games is all about. We don't want you to feel that you have to finish this book before trying out your own ideas—have a go whenever you feel ready, as you can always come back for more knowledge and ideas when you need them. Nonetheless, there is more to designing a good game than having a cool idea for a character or story, so these design chapters are here to provide some helpful advice for designing your own projects.

What Makes a Good Game?

We all know when we're playing one; we become completely absorbed by it and the hours fly by in no time at all, but how do you create a game like that? Well, unsurprisingly there's no formula to guarantee success—otherwise everyone would be doing it! However, there are some general principles that can help you to create better games by thinking more deeply about the way that games work. To become a good game designer, you need to learn to see beyond the surface features of games and consider what makes them fun at a basic level. This is something that takes time and practice, but we'll try to give you a taste of what we mean. Think about a particularly good section of your favorite game for a moment. Visualize where it is set, what your character is doing, and how it is interacting with the other characters and objects in the game. We're going to take fighting giant squids in *Zelda: Wind Waker* as an example:

> *The skies turn black and there is a crack of thunder as a giant squid rises from the surface of the ocean and towers over Link's tiny boat. A whirlpool forms around the monster's enormous body and the boat begins to circle helplessly around it in the current. The music reaches fever pitch and Link's only hope is to destroy all of the squid's bulging eyes with his boomerang before he is inevitably dragged down to a watery grave!*

Now this next part may seem a little strange: imagine that something has gone wrong with your PC or console and all the characters and objects in the game have turned into colored cubes! The music and sound effects have stopped too, but everything else is working just the way it always did. Now try to visualize your scene again:

A giant pink cube appears in front of my brown cube and I begin circling around it, gradually getting closer on each turn. There are eight white cubes attached to the pink cube—all of which must be destroyed before I get too close. To do this I must target white cubes in my line of sight and launch a yellow cube to fly out and destroy them. The yellow cube then returns back to me, and I can target and launch it again.

Now here's the question: would the cube version of the game still be fun to play? Well, it certainly won't be *as much* fun to play, since it has lost most of its original atmosphere and emotional involvement. However, for good games (like Zelda), some of the gameplay that makes it fun to play would still be there. It may look ridiculous—and you definitely wouldn't buy it like that—but part of the game's original magic remains.

Not quite convinced? Okay, take a look in the `Games/Chapter05` folder and play the example games, `evil_squares.gm6`, `galactic_squares.gm6`, and `lazarus_squares.gm6`. These are the three games you've already made but with simple shapes and sound effects, instead of the usual backgrounds, characters, and music. Give each game a chance and you'll soon see that there is still fun to be had once all the pretty graphics, characters, and stories have been completely stripped away (see Figure 5-1). Once you're convinced, read on . . .

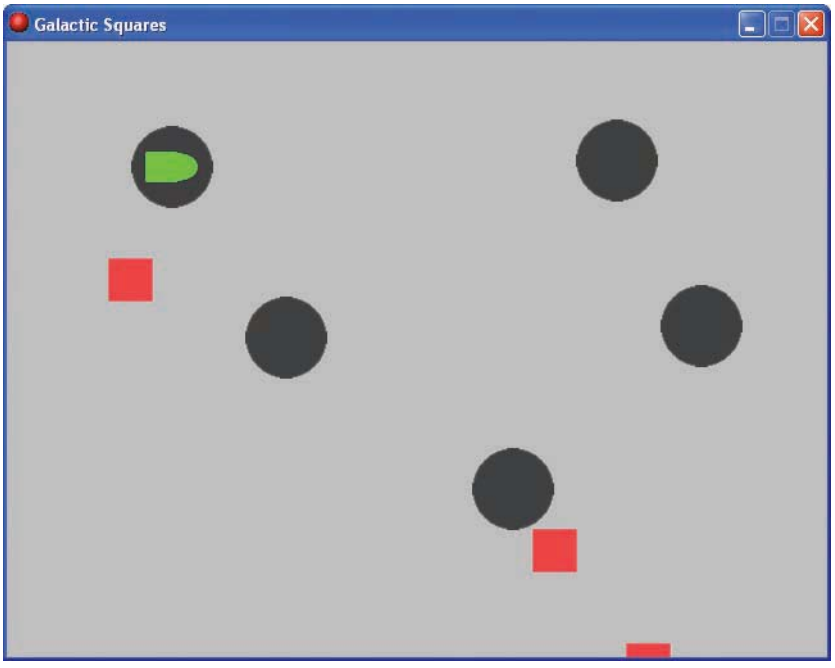

Figure 5-1. *Galactic Squares is not very pretty, but it's still fun to play.*

Game Mechanics

Okay, so good games are still playable even after all the fancy graphics and sound effects have been removed, but what creates this gameplay? Game developers call it the *game mechanics:* the basic rules and interactions that make a game fun to play. Understanding game mechanics

is probably the most important part of becoming a good game designer. Sure, creating appealing characters and stories is really important too, but they need to be combined with solid game mechanics to create a good game. Think of game mechanics as the engine of a car and the graphics, characters, and storyline as the bodywork and finishing. A rusty old wreck with a Formula One engine may not win the Grand Prix, but it stands more chance than a Formula One car with a rusty old engine!

Of course, the best games combine great game mechanics with superb graphics, believable characters, spectacular music, and compelling storylines. However, these other aspects are not unique to computer games, and there are plenty of books about filmmaking, storytelling, music, and artwork that cover these topics far better than we could. Therefore, these design chapters will focus on the core skill that distinguishes game designers from designers of other forms of entertainment: game mechanics.

Interactive Challenges

What's the difference between a film, a toy, and a game? It might sound like the start of a bad joke, but it's actually a question that highlights the two main features that make games special as a form of entertainment. The most obvious difference between films and games is that games are *interactive*—players have some control over the outcome of games, but film audiences do not. Toys (like train sets, for example) are also interactive, as players have control over what they want to happen when they play with them. However, toys don't provide *challenges* for the player in the same way that games do. A player can create their own challenges using a toy (like deciding to race trains), but those challenges have to be created by the player and are not part of the toy. A game normally comes with its own set of challenges that the player must overcome in order to win the game.

So you can think of games as being "interactive challenges," therefore it's easy to deduce that both interaction and challenge are key elements of game mechanics. For the remainder of this chapter, we'll look at the various ways these two elements improve the game mechanics of your designs and make them more fun to play.

Game Genres

We often group games into different *genres*, and one way of doing this is to look at the types of interactions and challenges that different kinds of games provide. Games are evolving all the time, so there will never be a final list of genres that everyone agrees on. Nonetheless, we have made our own list of the main genres. As you read each one, try to distinguish the role of the game mechanics from the part that the characters, stories, and graphics play in the experience of that genre.

- **Action games** (e.g., sports, combat, platform, racing) usually involve fast and furious interactions with lots of physical challenges that leave little room for mistakes.
- **Simulator games** (e.g., flight sims, racing sims) usually involve realistic interactions and physical challenges with no room for mistakes at all.
- **Strategy games** (e.g., war games, puzzle games, god games) often involve slow or turn-based interactions with long-term intellectual challenges that involve planning and organization.

- **Adventure games** (e.g., point-and-click) usually let players interact at their own pace, providing short-term puzzle-based challenges and long-term story-led challenges. These challenges are often impossible to fail if you keep trying.
- **Role-playing games** (e.g., online RPGs) usually provide slower interactions with long-term story-led challenges. However, these are often less important to the player than the story and challenges that the player creates for themselves while developing their character.

Of course, most games don't fall neatly into one genre and may combine several kinds of interactions in one game. Nonetheless, a game designer does need to consider players' expectations of a particular genre; a role-playing game that requires lightning reflexes or a turn-based shoot-em-up might not go down too well! It's also worth remembering that new genres are only created when rules and conventions are broken, and the great games of the future are unlikely to follow the same conventions as today.

Challenges

We hope that you can see from the game genre descriptions that different players want different kinds of challenges from the games they play. Despite this, there are some general guidelines that can help you to provide better challenges in your games. We're going to apply these guidelines to the Evil Clutches game from Chapter 2 to see if we can turn it into a better game. All the new versions of the game can be found in the `Games/Chapter05` folder on the CD. We've provided these as `.exe` files because we just want you to play them and notice how the changes are affecting the gameplay. You really don't need to know how they are made, but you can find the corresponding Game Maker project files in the `Games/Chapter05/Registered` directory on the CD. However, because these versions contain effects that are only available in the registered version of Game Maker, you will need to use the executables to play the game if your copy is still unregistered.

Difficulty

Challenges are important in games, because beating challenges makes players feel good about themselves. For this to happen, a challenge must be easy enough for a player to achieve but hard enough to be worth bothering with. Players give up on games that are too easy, because there is no satisfaction from beating a challenge that you could do blindfolded. However, players give up on games that are too hard because it makes them feel bad about themselves for failing, and they don't feel they are making any progress.

At the moment our Evil Clutches game is far too difficult at the start of the game, but in other ways it's too easy as well. It is too hard because just one touch of a demon will kill the dragon, and the game can be over before the player has worked out the controls! However, it can become too easy later on if players realize that they can always safely hide just offscreen and swoop in to rescue the hatchlings.

Even once these issues have been fixed, the game will still be too difficult for some players and too easy for others. People have different amounts of experience with computer games, but the best games are the ones that players of all levels can get into. We're going to make sure that our game appeals to as many people as possible by adding a difficulty menu at the start of

the game, allowing the player to play in easy, medium, or hard mode. So in combination with the other tweaks, these are the changes that we're going to make to the first version of our game:

- Display a health bar for the dragon starting with 100 points of health.
- Make the dragon lose only 10 health points for each collision with a demon.
- Prevent the dragon from leaving the screen.
- Add a difficulty menu at the start of the game for easy, medium, or hard mode.

The file `evil_new1.exe` contains these four changes to the game. Play the new version and see what you think. We've changed the difficulty of the game by adjusting the chance of demons and hatchlings appearing in the different modes. There are now extra demons and fewer hatchlings in the hard mode and fewer demons and extra hatchlings in the easy mode. When you're setting the difficulty of your games, remember that game developers always find their own games easier than anyone else because they play them so much. If you make your own games harder and harder as you get better and better at them, then they will end up too difficult for other players. Always get someone else to test your game to make sure you've got the difficulty levels about right, and if you can't complete the game yourself, then don't expect anyone else to be able to!

Goals

Challenges are created by setting clear goals for players to achieve. If a goal is unclear or forgotten, then it no longer creates a challenge and it loses its power. In the last version of the game, we sneaked in some extra actions that made saving the lives of 50 hatchlings the ultimate goal of the game. However, you won't have felt any more challenged, since you didn't know about this new goal! In fact, even once you know about it, it's difficult to keep track of how many hatchlings you've saved, so any interest in the challenge doesn't last very long. It may sound obvious, but to keep a player challenged you need to make sure that they know what their goals are, and how they are progressing with them. Our game currently has two main goals for the player: saving a set number of hatchlings and beating the top score on the high-score table. We can make sure that these goals challenge the player by clearly displaying information about the player's progression toward these goals on the screen.

When players know what their goals are and how close they are to completing them, it also begins to create the what-if effect when the player loses. The closer a player gets to their goal, the more likely they are to think, "What if I had just been a little bit quicker, or hadn't made that one, stupid mistake?" Of course, this only happens if players get close to their goals—if they don't make it past the first obstacle, then they are more likely to think "As if!" than "What if?" This means that a player who rescues 40 of 50 hatchlings is more likely to have another go than a player who only rescues 10 of 50 hatchlings. We could make our game so easy that everyone can rescue 40 hatchlings, but then the game would become too easy to complete and lose its challenge. Instead, we can be more devious and reduce the chance of hatchlings appearing as more hatchlings are rescued. So if the chance of the first hatchling appearing is 1 in 50 (a 50-sided die), then the second might be 1 in 52, the third 1 in 54, and so forth. That way, by the time the last hatchling appears, the chance of the hatchling appearing has changed to only 1 in 150 (a 150-sided die!). In practice, this just means that hatchlings are

released more quickly at the start of the game than at the end. This will make it easier for all players to rescue a good number of hatchlings—and trigger the what-if effect—without making the game too easy overall.

So to incorporate all these improvements to our game's goals, we'll make the following changes to the second version of the game:

- Clearly display how many hatchlings have been rescued and how many need to be rescued in total.
- Clearly display the player's score and the top high score to beat from the high-score table.
- Reduce the chance of hatchlings appearing, based on the number that have already been rescued.

Play the file `evil_new2.exe` containing these three changes. We hope you'll agree that we're already starting to make progress toward a game that is much more fun to play than the original (see Figure 5-2).

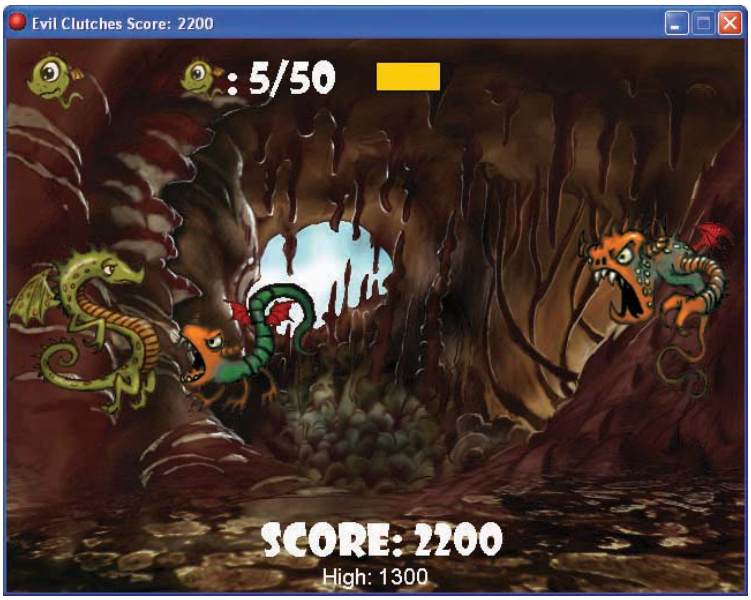

Figure 5-2. *Finally we can see our goals and how far away we are from achieving them.*

Rewards

Rewards are extremely important for maintaining a player's interest in a game's challenges. It can take a lot of time and effort to complete a challenge, so a reward makes players feel much better about it. It also makes it much more likely that they'll want to complete other challenges offered by the game. The high-score table already provides a reward system for scores, but we could do with something extra special for reaching the end of the game. So, once all the hatchlings have been rescued, we will:

- Display a congratulatory message.
- Award the player with a large bonus score.
- Show the player an amusing conclusion to the story of the game.

Although it is most important to reward players for completing the game's goals, it also helps to occasionally give them small rewards for no reason at all. Games often do this in the form of health bonuses and other kinds of pickups, which appear at random intervals. The fact that they appear randomly is significant, as it gives players hope that a pickup may come along at any point. This means they are more likely to stick with the game when they're in a desperate situation where they might otherwise give up—and if a bonus does arrive just in time, then the feeling of relief is enormous. It also adds to the power of the what-if effect, as players can now think, "What if I'd had just one more health bonus—maybe I'll be luckier next time?" We're going to add our own random rewards to Evil Clutches by making the following changes:

- Make the boss demon randomly drop health and shield bonuses.
- Randomly add between 5 and 25 percent to the dragon's health when a health bonus is collected.
- Make the dragon immune to taking damage for 15 seconds when a shield bonus is collected.

All of these new rewards are included in the file `evil_new3.exe`. The animation at the end of the game is an example of the kind of animated rewards you can quickly create in Game Maker with a little bit of imagination. It's not exactly a beautifully rendered cut-scene (see Figure 5-3), but it should make players smile.

Figure 5-3. *If you mess with dragons, then you're bound to get your fingers burned!*

Subgoals

Subgoals can enhance your games by providing short-term or optional challenges for your players to take up. Most games include a long-term goal that must to be met in order to complete the game, but these can often seem very distant and hard to achieve at the start of the game. Subgoals give the player something to aim for in the short term, and good games tend to provide a series of both short- and long-term goals to draw the player through the game. Our game is not very long, but there is certainly room for an additional short-term challenge. We'll challenge the player to shoot demons without taking damage, and reward them by powering up their fireballs as their demon tally increases. To achieve this we will:

- Count and display the number of demons shot in a row and reset the count back to 0 when the dragon takes damage.
- Limit the number of fireballs in the air at once, based on the current demon count. Begin with a limit of 2 and add 1 to this for each 10 demons on the tally.
- Scale the size of the fireball and add smoke effects to make the fireballs look more impressive as the demon tally increases.

Optional subgoals are a good way of providing extra challenges to advanced players, which other players can choose to ignore. These often include collecting particular items to unlock extra options, or hidden levels that less adventurous players are unlikely to find. These are really just a different way of balancing the difficulty of your game, so that players naturally find the right level of challenge for their own abilities. For our game we're going to turn the collection idea on its head and add a subgoal of trying not to accidentally shoot hatchlings! To make this work, we'll include the following changes:

- Each time a hatchling is accidentally shot, subtract one from the total number of hatchlings that have to be rescued (already displayed).
- At the end of the game, award the player bonus points based on the total number of hatchlings they've saved.

Ideally this should have its own special reward at the end of the game, but to keep it simple we've just rewarded the player handsomely in points for each hatchling saved. You can play a version of the game with these new subgoals in the file `evil_new4.exe`.

Interactivity

As well as challenges, the other main feature of games is their interactivity. Interactivity is about putting the player in control. Good games leave the player feeling in control of the game, while bad games make them feel powerless. As with challenges, players of different game genres often prefer different levels of control, but there are some common ways of helping to maintain a feeling of control in your games.

Choices and Control

To give players a feeling of control, we need to provide them with choices that seem to have a real effect on the outcome of the game. Action games constantly require players to make choices about the physical actions of the game (jumping, shooting, flying, etc.) and so provide an immediate feeling of control. However, games of all genres should ensure that enough choices are available to create this feeling too. Adventure games without enough choices can seem very linear—as if you are being forced through a path that has already been decided for you. Whenever you add choices to your games, think carefully about the difference they really make: is it worth having ten different weapons that all work in the same way? What's the point in allowing the player to choose what to say to a character if it always has the same outcome? Adding these kinds of features won't generally make your game any worse, but changing them so that they make a real difference will give more control to your players and make them more involved in the game.

We're going to add a choice of characters to our game, so that the player can choose to play the mother dragon or the father dragon. To make sure that this is a choice worth making, we're going to give each character a different special ability, which affects the way they play. The mother dragon will be able to call hatchlings to her—causing them to speed up and get out of harm's way more quickly—and the father dragon will be able to blast demons at close quarters with a cloud of steam—sending them crashing back into other demons. To add the new character and include different separate abilities, we will:

- Add the ability to select between playing the mother and father dragons at the start of the game.
- Add the ability for dragons to activate their special abilities using the Ctrl key.
- Make hatchlings at the same vertical level as the mother dragon speed up when she calls them to her.
- Make demons close to the father dragon fly back into other demons when he blasts them with a cloud of steam.

Figure 5-4 shows the character selection screen. You can see the effect of all these new changes by playing the file `evil_new5.exe`.

Control Overload!

Of course, it is possible to have too many choices in a game—particularly if extra choices mean extra controls. Most people can remember between five and nine things at once. If you have more than five controls in your game, some players will have forgotten what the first key does by the time they read what the last key does. In general, it's probably not a good idea to have more than two controls plus the arrow keys to move around. Try to make controls automatically perform different functions depending on the situation: pressing the spacebar might pick up items, open doors, or attack creatures, depending on whether the player was near to an item, a door, or a creature. That way, you can include lots of interesting features without needing extra controls for the player to remember.

Fortunately, this is one area where our Evil Clutches game is okay. Some players may find the special move control a bit too much to cope with at first, but because these moves are optional, we don't need to worry about them too much.

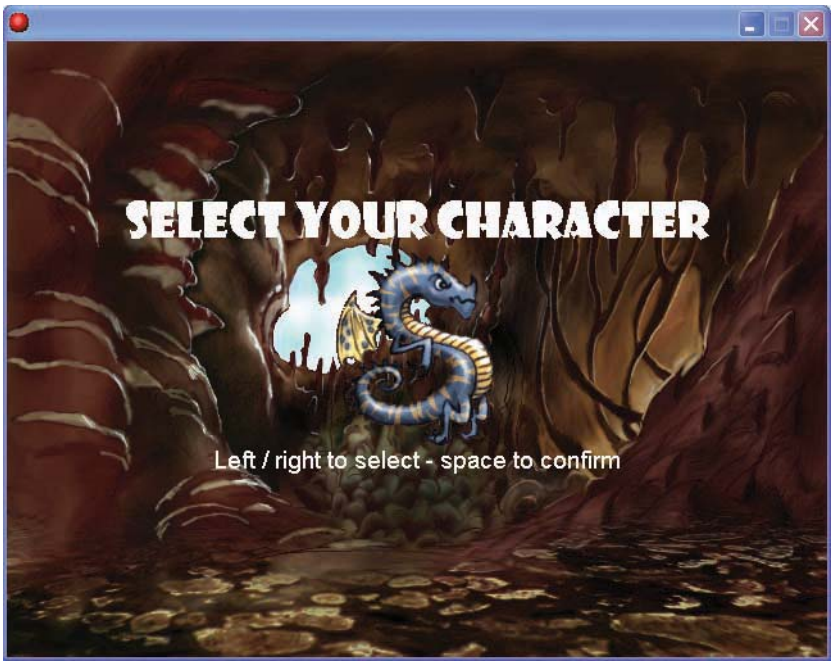

Figure 5-4. *It's one big happy family—and he's the daddy!*

Unfair Punishment

With the right level of control in your games, players will feel that they are the makers of their own fortunes. However, you can still quickly convince them otherwise by punishing them for something that isn't under their control. Such punishments are usually not included intentionally, but friendly characters with suicidal habits and enemies that blow up your objectives are both examples that have accidentally made it into commercial games. Avoiding unfair punishment is usually about making sure your game still works correctly, even when a player isn't playing the game in exactly the way you intended. The best way to find these problems is to get your friends to test your game thoroughly. They'll soon tell you if they think that something is unfair about your game.

Our game occasionally punishes the player unfairly. Fireballs go straight through demons, so it's easy to accidentally kill a hatchling that is flying behind one. The hatchling may not have even appeared until after the player pressed fire, but it is too late for them to do anything about it. When this happens the player may feel frustrated at being punished for something that was out of their control. We'll solve this problem by making the fireballs disappear when they kill a demon. This also has the effect of making the game a bit more fast and frantic, which is not a bad thing for a shoot-em-up.

Audio Feedback

In the final version of Evil Clutches, we're also going to add more sound effects to improve the game mechanics. At first glance, this may seem to go against the idea that mechanics are about rules and mechanisms—not niceties like sound. However, sounds are not just included in games just because they "sound nice," but also because they provide useful feedback to the player about what they are doing. If you go back to the Galactic Squares example from the start of the chapter, you'll notice that it still includes very basic sound effects. These sounds are designed to quickly inform the player about whether their interactions with the game are good or bad. Audio designed in this way can play an important part in helping to naturally steer the player in the right direction, whereas otherwise they might end up confused. Confused players do not feel in control, so audio has a role to play in this too.

Designing sound effects that both inspire the senses and inform the player in this way is not easy, and commercial games have their sound effects designed by professional sound engineers. See Chapter 15 for more information on the kinds of tools that you can use to try to do this for yourself. You'll find our sound effects in the `Resources/Chapter05` folder and can hear them in action by playing `evil_new6.exe` from the `Games/Chapter05` folder. This final version of the game includes the following changes to the unfair punishment and audio:

- Make fireballs disappear when they collide with a demon.
* Add audio feedback for shooting fireballs.
- Add audio feedback for when the dragon takes damage.
- Add audio feedback when a hatchling is saved.
- Add audio feedback for pickups.
- Add audio feedback for menus.

Summary

Now that you've played the final version of the game, we hope you found it much more fun than the original from the end of Chapter 2. In this chapter we've learned that challenges and interactivity are a central part of the game mechanics that make games fun to play. We've looked at a number of general principles that can help you to create better interactive challenges and followed them through with the Evil Clutches example. These are certainly not the only principles of good game design, and you are unlikely to design a good game simply by following a set of rules. Nonetheless, here is a summary of the main issues as a starting point for your own Game Maker projects:

- Challenge the player by
  - Providing clear, achievable goals and giving feedback on the player's progress.
  - Including both long and short-term goals.
  - Adding difficulty levels and optional subgoals for players of different abilities.
- Reward the player
  - For achieving goals and subgoals.
  - Randomly.
- Make the player feel in control by
  - Giving them choices that seem to make a real difference to the game.
  - Not confusing them with too many controls.
  - Not punishing them for things out of their control.
  - Giving the player audio feedback about their interactions with the game.

If you apply these principles with a bit of thought and care, then you should find that they can help you to make your own games more fun to play too. That concludes this chapter and the second part of the book—we'll look at some more design principles at the end of the next part, but for now we're joining the creatures of a Japanese coral reef to learn something about parenting . . .

PART 3

Level Design

The quality of a game's level design can make or break a game. Your game's popularity will sink like a brick if you don't put enough time into getting it right!

CHAPTER 6

Inheriting Events: Mother of Pearl

These days, some of the most inventive games come out of Japan. Japanese designers have a history of taking crazy design scenarios and turning them into brilliantly addictive games (e.g., Puzzle-Bobble, Pikmin, Gitaroo Man). Japanese games also have their own distinctive look derived from manga comics. In this chapter we'll make our own game in this style, based around the classic game of Breakout. In doing so, we'll learn how to use parent objects, one of Game Maker's most powerful features. As always, though, we'll need to start by writing a quick description of our game design.

Designing the Game: Super Rainbow Reef

Titles for Japanese games often start with the word "Super" (Super Monkey Ball, Super Smash Brothers, Super Street Fighter, etc.), so we're calling this game *Super Rainbow Reef* to keep in with the theme. Here's the design:

The monstrous Biglegs have driven the peace-loving creatures of Rainbow Reef from their ancestral homes. Despite their inexperience in the ways of war, Pop and Katch have invented a way of combining their skills to fight back against the Biglegs. For this incredible feat, Pop must bounce from Katch's shell to attack the evil invaders. Katch must then move quickly to save Pop from plummeting into the deep waters below. The cowardly Biglegs often retreat behind coral defenses, so our heroes must be prepared to smash their way through if they are to finally drive the Biglegs from Rainbow Reef!

There will be no direct control over Pop's movement, and he'll bounce freely around a playing area enclosed by walls on all sides except the base. The left and right arrow keys will move Katch horizontally along the base in order to bounce Pop from Katch's shell and stop him from falling out of the level. The collision point along Katch's shell will determine the direction of Pop's bounce, and so allow the player to control his movement. Bounces toward the left will send Pop left and bounces toward the right will send him right. Pop's movement is also affected by gravity, and each time he collides with Katch, he gets slightly faster so that the game becomes increasingly difficult.

The game will have several levels, each containing a number of Biglegs that Pop must collide with in order to complete the level. Most levels will also contain coral block defenses, which must be knocked out of the way in order to reach the Biglegs. Breaking blocks will score extra points and special blocks give the player extra rewards, but they don't have to be destroyed to

finish a level. If Pop leaves the screen, the player loses a life and Pop is brought back into play. Once three lives have been lost, the game ends and a high-score table is displayed. A typical level is shown in Figure 6-1.

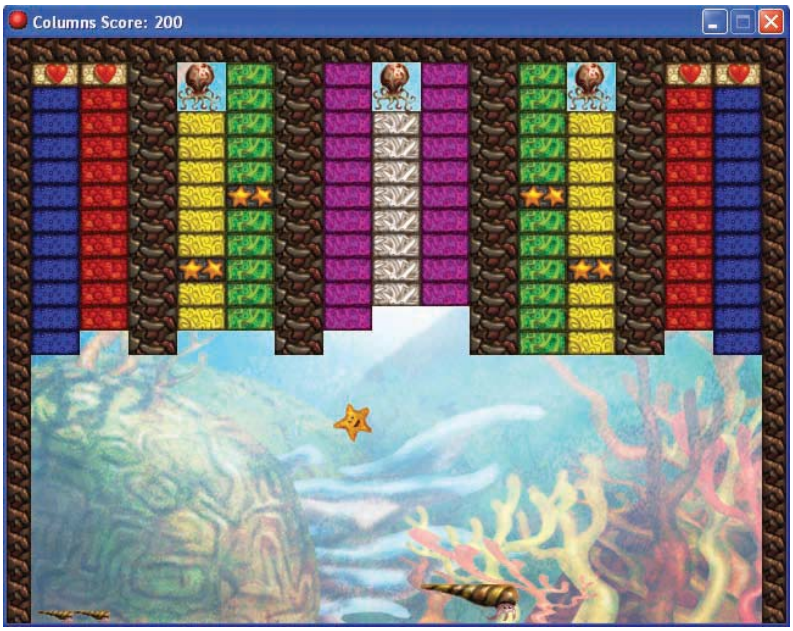

Figure 6-1. *Here's a typical level in Super Rainbow Reef.*

This time we will also make a separate feature list. It consists of all the different kinds of enemies and blocks that are available to create each level:

- Enemies:
  - Large stationary Biglegs
  - Small stationary Biglegs
  - Large Biglegs that move horizontally
  - Small Biglegs that move horizontally
- Blocks:
  - Multicolored blocks that can be destroyed for points
  - Solid blocks that cannot be destroyed
  - Blocks that must be hit twice before they are destroyed
  - Invisible solid blocks that cannot be destroyed
  - Blocks that create two extra copies of Pop when destroyed
  - Blocks that give the player an extra life

This should give enough information to create the game. All resources for this game have already been created for you in the `Resources/Chapter06` folder on the CD.

A Game Framework

From now on, we are going to use a standard framework for many of the games we create so that they all have start and end screens that work in the same way. This framework will have the following parts:

- A title screen that displays the name of the game and has buttons to start a new game, load a saved game, show help, display the high-score table, and quit the game. Game developers normally use the term *front-end* to refer to the part of the game that allows the player to make selections like this. This screen will also be responsible for starting the background music and for initializing any other game settings, like the score.
- The actual game.
- The end of game screen that is shown when the game has been completed. This displays a congratulatory message and activates the high-score table, after which the game returns to the front-end.

You should already be familiar with how to do most of this from previous chapters, but this time we're adding button objects so that the player can control what is going on.

The Front-End

We'll begin by creating the front-end. Commercial games often have developers who devote all their time to creating the front-end for a game. Fortunately, ours won't be that complicated; all we need is a background image, a title sprite, several button sprites, and some background music. Let's start by adding these to the game.

Creating the front-end resources for the game:

1. Launch Game Maker if you haven't already and start a new empty game.
2. Create a new sprite called `spr_title` using `Title.bmp` from the `Resources/Chapter06` folder on the CD (you can use `Title.gif` instead if you prefer the look of it).
3. Create the five button sprites—`spr_button_start`, `spr_button_load`, `spr_button_help`, `spr_button_scores`, and `spr_button_quit`—using the appropriate sprites from the same directory. Disable the **Transparent** option on these sprites.
4. Create a new background called `background1` using `Background1.bmp` from the `Resources/Chapter06` folder.
5. Create a new sound resource called `snd_music` using `Music.mp3` from the `Resources/Chapter06` folder.

This gives us all the resources we need for creating the front-end, so now we can define the objects for it. We'll start with the object that displays the title sprite, sets the score to 0, and

plays the background music (we'll give this object some other functions as well later on). After that we'll create the button objects for the front-end.

Creating the title object resource for the front-end:

1. Create a new object called `obj_title` and give it the title sprite.

2. Add a **Create** event to the object and include a **Set Score** action to set **Score** to `0`.

3. Add an **Other, Game Start** event and include a **Play Sound** action (**main1** tab) in it. Select the background music, and set **Loop** to true so that the music plays forever.

4. Click **OK** to close the properties form.

Creating the button objects resources for the front-end:

1. Create a new object called `obj_butstart` and give it the start button sprite.

2. Add a **Mouse, Left Pressed** event and include the **Next Room** action (**main1** tab). The **Left Pressed** event happens when the user clicks on an instance's screen position with the left mouse button.

3. Click **OK** to close the Object Properties form.

4. Create a new object called `obj_butload` and give it the load button sprite.

5. Add the **Mouse, Left Pressed** event and include the **Load Game** action (**main2** tab). **File Name** must be the same name as the file that we use to save the game to later on, so it's easiest to leave this as the default setting.

6. Click **OK** to close the Object Properties form.

7. Create a new object called `obj_buthelp` and give it the help button sprite.

8. Add the **Mouse, Left Pressed** event and include the **Show Info** action (**main2** tab). This action shows the player the text entered under Game Information.

9. Click **OK** to close the Object Properties form.

10. Create a fourth button object called `obj_butscores` and give it the score button sprite.

11. Add the **Mouse**, **Left Pressed** event and include the **Show Highscore** action (**score** tab). Feel free to play around with the settings to make the table look nice.

12. Click **OK** to close the Object Properties form.

13. Create a final object called `obj_butquit` and give it the quit button sprite.

14. Add the **Mouse**, **Left Pressed** event and include the **End Game** action (**main2** tab).

15. Click **OK** to close the Object Properties form.

This gives us all the objects we need. Now we need to create a room to put them in that acts as the front-end itself.

Creating the front-end room resource for the game:

1. Create a new room resource called `room_frontend` (**settings** tab). Give the room a caption, then scale the room window so that you can see as much of the room as possible (preferably all of it).
2. Switch to the **backgrounds** tab. Click the menu icon to the right of where it says **<no background>** and select the background from the pop-up menu.
3. Next switch to the **objects** tab. Place one instance of each of the button objects along the bottom of the room (or somewhere else if you prefer). A logical order would be Start, Load, Help, Scores, Quit.
4. Select the title object and position it in the center of the room (remember that you can move an instance by holding the Ctrl key). Your room should now look something like Figure 6-2.

■ **Note** When you add an instance on top of an existing instance, the existing instance is deleted. This can be avoided by disabling the **Delete Underlying** option at the bottom left.

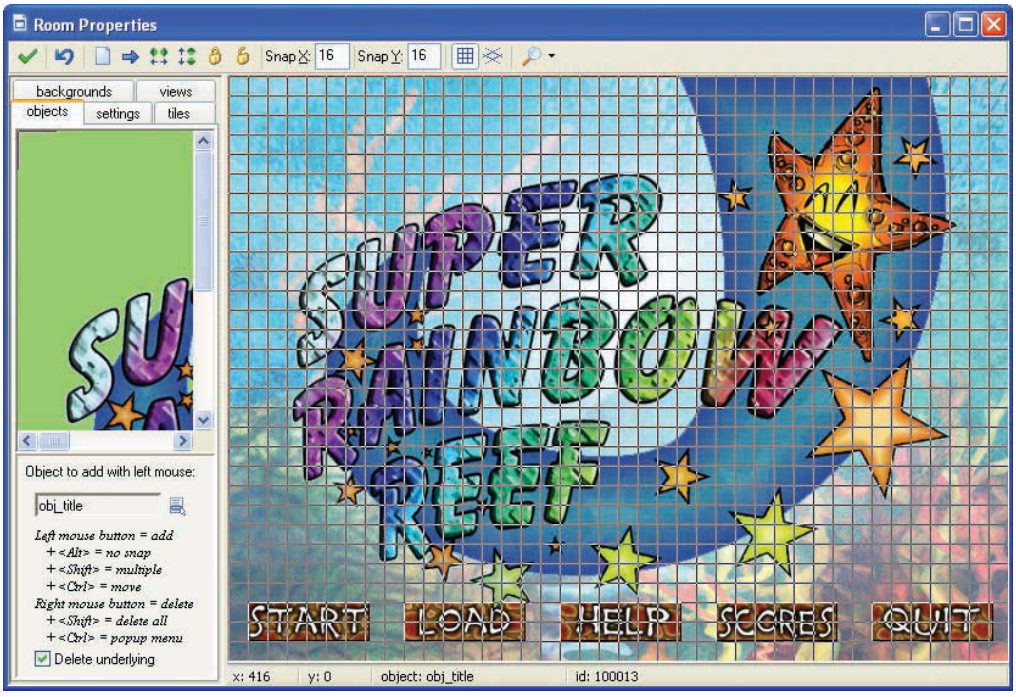

Figure 6-2. *The front-end room looks like this in the room editor.*

Finally, you should write a short help file for the game. You may find this easier once we have finished making the game, but we'll remind you how to do it now anyway.

Adding game information to the game:

1. Double-click on **Game Information** near the bottom of the resource list.
2. Type the name of the game, your name as the creator, a short description of the objectives, and a description of the controls.
3. Click on the green checkmark at the top left to close the editor.

You might want to test the framework now to check that it works correctly. An error message will appear if you click the start button, but that's fine as there are no rooms to go to yet in the game. Likewise, the load button will not do anything as there is no saved game to load, but the help, scores, and quit buttons should all work correctly.

The Completion Screen

We'll create the completion screen in a similar way to the one described in the previous chapter. It will contain one object that displays some text congratulating the player and adds 1,000 points to the player's score. There will then be a short pause and the high-score table will be displayed. After this, the player returns to the opening screen.

To achieve this, we're going to use new kinds of events and actions called *alarms*. *Alarm actions* work like a countdown for triggering *alarm events*. Alarm actions are given an amount of time to wait, and an alarm event is executed at the moment that time runs out. Like in other places in Game Maker, this time is measured in steps, so for each alarm action you indicate how many steps there should be before the alarm event takes place. Each instance of an object can have up to 12 alarms that can be used to make things happen at set points in the future.

Creating a congratulation object that uses alarm events and actions:

1. Create a new sprite called `spr_congrats` using the file `Congratulation.gif`.
2. Create a new object called `obj_congrats` and give it the congratulation sprite.

3. Add a **Create** event and include the **Set Alarm** action (**main2** tab). Set **Number of Steps** to `120` to create a delay of 4 seconds (as there are 30 steps in a second). Leave **In Alarm No** set to Alarm 0. See Figure 6-3.

4. Include a **Set Score** action below the alarm using a **New Score** of `1000`, and enable **Relative** so that the score is added on.

5. Add an **Alarm, Alarm 0** event. This is the event that will now happen four seconds after the alarm action is executed. Include a **Show Highscore** action in this event and give it the same background and font settings as before.

6. Include the **Different Room** action (**main1** tab) and select the front-end room.
7. Click **OK** to close the Object Properties form.

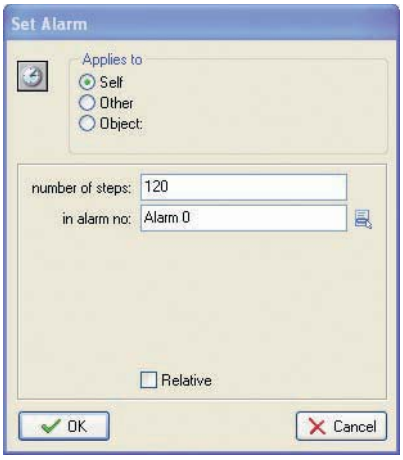

Figure 6-3. *Set an alarm clock to 120 steps, which is equivalent to four seconds.*

Creating a completion room for the game:

1. Create a new room called `room_completed` (**settings** tab) and give it an appropriate caption.
2. Switch to the **backgrounds** tab. Click the menu icon to the right of where it says **<no background>** and select the background from the pop-up menu.
3. Switch to the **objects** tab. Select the congrats object and position it in the top-left corner of the room.

It is time to test your work and try all the buttons again. Playing the game should now take you straight to the closing screen (you've finished all the levels as there are none!). You can now also add your name to the high-score table and view it again from the front-end. If any of this isn't working correctly, then carefully check that you've followed the instructions correctly. Alternatively, you can load this version from `Games/Chapter06/rainbow1.gm6` on the CD.

We'll use this same framework (with different graphics) for many of the remaining games in the book. We won't have the space to explain all the steps again in so much detail, so make sure that you understand it well before continuing. Otherwise, you can either refer back to this chapter, or simply load each game with the framework already made to save yourself some work.

Bouncing Starfish

Now that our framework is complete, let's get started on the fun part: developing our game. In this section, we will create Pop, Katch, and the level boundary so that we can get Pop bouncing around the screen. As usual, we begin by creating the relevant sprites for the game.

Creating new sprite resources for Pop, Katch, and the wall:

1. Create a sprite called `spr_wall` using `Wall.gif` from the `Resources/Chapter06` folder on the CD. Disable the **Transparent** option so that the wall appears completely solid.
2. Create a sprite called `spr_pop` using `Pop.gif`. Click the **Center** button to place the origin of the sprite (where it is "held" from) at its center. Moving Pop from his center will make it easier to work out how far he has landed along Katch's shell when they collide. Also enable the **Smooth Edges** option to make the edges of Pop's legs look neater.
3. Create a sprite called `spr_katch` using `Katch.gif`. Click the **Center** button again and enable the **Smooth Edges** option too.

The next step is to create the corresponding objects. The wall object is extremely easy: it doesn't do much except act as a solid boundary for the playing area.

Creating the wall object resource for the game:

1. Create a new object called `obj_wall` and select the wall sprite.
2. Enable the **Solid** option and click **OK** to close the form.

Next we will add an object for Katch. Although Pop and Katch are both on the player's team, only Katch can be directly controlled by the player. The left and right arrow keys need to move Katch in the appropriate direction when there are no walls in the way. We will check for walls using a **Check Object** action that tells us if another kind of object is nearby. By making this check before moving Katch left or right, we can make sure that we only move her when her path is not blocked.

Creating the Katch object resource for the game:

1. Create a new object called `obj_katch` and give it Katch's sprite.
2. Add a **Keyboard, <Left>** event.
3.  Include the **Check Object** conditional action (**control** tab) in this event. Select the wall object, set **X** to `-10` and **Y** to `0`, and enable the **Relative** option. This will check for wall objects 10 pixels to the left of Katch. Also enable the **NOT** option. This reverses the condition so that the next action (to move Katch) will be executed only if there is *not* a wall object in the way. The action should now look like Figure 6-4.
4.  Include the **Jump to Position** action (**move** tab). Set the same values for **X** and **Y** (`-10` and `0`) and enable the **Relative** option so that Katch moves into the position that we just checked was free of walls.
5. Add a **Keyboard, <Right>** event.
6.  Include the **Check Object** action (**control** tab). Select the wall object, set **X** to `10` and **Y** to `0`, and enable the **Relative** option. Also enable the **NOT** option again. This performs the same check for walls to the right.
7.  Include the **Jump to Position** action (**move** tab). Set **X** to `10` and **Y** to `0`, and enable the **Relative** option. This moves Katch to the right if there are no walls blocking the way.

■ **Note** All conditional actions have a **NOT** field. This is used to indicate that the following actions are executed only if the condition is ***not*** true.

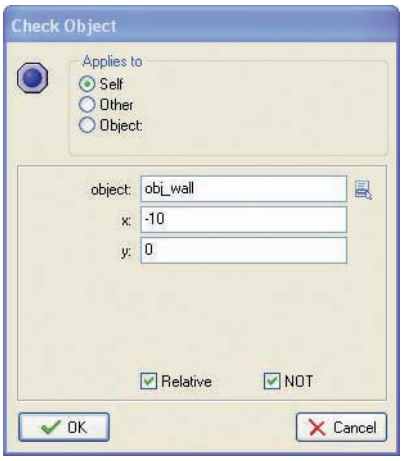

Figure 6-4. *This action checks that there is no wall object to the left of Katch.*

■ **Tip** When there is just one action that must be executed as a result of a condition, there is no need to put start and end blocks around it.

That completes the Katch object; now we need to create an object for Pop. This will be more complicated as it needs to move around and bounce against the walls. We'll also add some gravity so that Pop moves more realistically, and floats back down toward Katch at the bottom of the screen.

Creating the Pop object resource for the game:

1. Create a new object called `obj_pop` and give it Pop's sprite. Set its **Depth** to `10` so that it appears behind other objects in the game (this will look better later on).

2. Add a **Create** event and include a **Move Free** action with a **Speed** of `12`. We want Pop to start moving upward, but we'll add a little variation by typing `random(60)+60` as the **Direction**. The `random(60)` part will produce a random number between 0 and 60, so by adding 60 to this we will get a value between 60 and 120. This means we will get a direction somewhere between the two green arrows in Figure 6-5.

Note Angles in Game Maker work slightly differently from the way you might be used to. There are still 360 degrees in a full circle, but 0 degrees is horizontally to the right and they increase in an anticlockwise direction (see Figure 6-5).

3. Include a **Set Gravity** action (**move** tab), setting **Direction** to `270` (directly downward, see Figure 6-5) and **Gravity** to `0.2`. This means that a downward speed of 0.2 will be added to Pop's current speed in each game step, pulling him slowly toward the bottom of the screen. The form should now look like Figure 6-6.

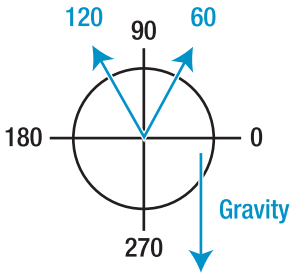

Figure 6-5. *Angles in Game Maker work differently from the way you may be used to.*

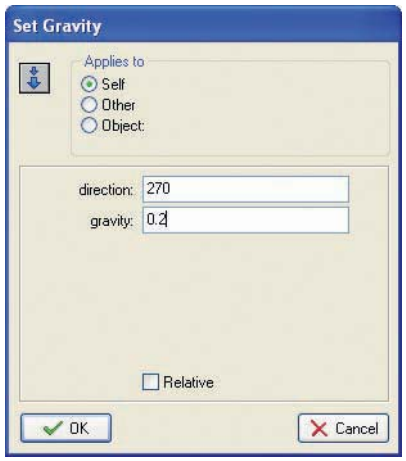

Figure 6-6. *This action sets the gravity downward by using an angle of 270 degrees.*

4. Add a **Collision** event with the wall object and include a **Bounce** action. Set the **Precise** option to **precisely** so that it takes into account the exact appearance of the colliding sprites to calculate the result of the collision.

Now we need to create actions to make Pop bounce off Katch's shell. We could use the **Bounce** action again, but then Pop would always bounce in the same way. We want the player

to be able to control the direction of the bounce, so we need to alter it according to how far Pop bounces along Katch's shell. A bounce in the center of the shell should send Pop straight up the screen, but a bounce toward the end will send Pop diagonally in that direction (see Figure 6-7). In fact, we're going to send Pop bouncing off at an angle between 50 and 130 degrees, depending on his collision position.

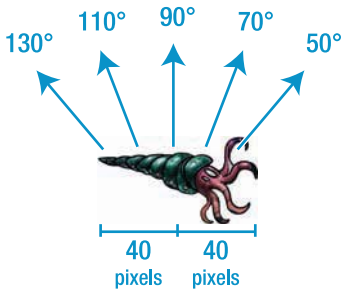

Figure 6-7. *The angle that Pop bounces will depend on how far along Katch's shell he collides.*

To achieve this, we need to compare Pop and Katch's horizontal positions at the moment they collide. Remember that both Pop and Katch's sprites have their origins in the center, so when their horizontal (x) positions are equal, then Pop is exactly in the middle of Katch's shell. When this is the case, we want Pop to rebound with an angle of 90 degrees, straight upward.

However, if Pop's x-position is smaller than Katch's x-position, then Pop has landed on the left side of the shell, which means he should bounce more to the left—with an angle of more than 90 degrees. Similarly, if Pop's x-position is larger than Katch's, then he has landed on the right and we need a direction smaller than 90 degrees. You should remember from previous games that we can get an object's own x-position using the `x` variable, and the x-position of another object (like Katch) by putting its name in front of it like this: `obj_katch.x`.

So we can work out the difference between Pop's and Katch's horizontal positions by subtracting one from the other: `obj_katch.x-x`. This difference will be a positive number if Pop lands on the left side of Katch and a negative number if he lands on the right (try working it out on paper for a few example positions, if it helps). It will also be exactly zero when Pop is in the middle. So actually, all we need to do is add 90 to this difference (`90+obj_katch.x-x`) and it will give us the range of angles between 50 and 130 degrees we're after (see Figure 6-7). We'll also make the game get harder over time by increasing Pop's speed each time he collides with Katch.

Adding a collision event to the Pop object for colliding with the Katch object:

1. Add a **Collision** event with the Katch object and include a **Move Free** action in it. Type `90+obj_katch.x-x` in **Direction** and `speed+0.3` in **Speed** (this adds 0.3 to the current speed).

2. Finally we need to restart the room when Pop falls off the bottom of the screen. Add the **Other, Outside Room** event and include the **Restart Room** action (**main1** tab).

These are all the objects we need to start testing our game, so let's quickly create a test room to do this. This room must be inserted between the front-end room and the completed room in the resource list.

Creating a new test room resource for the game:

1. Right-click on `room_completed` in the resource list and select **Insert Room** from the pop-up menu that appears. This will insert a new room before `room_completed` and open the properties window for the new room.
2. Switch to the **settings** tab and call the room `room_test`.
3. Switch to the **backgrounds** tab and set the background for the room.
4. Our wall sprites are 20 by 20 pixels, so set both **Snap X** and **Snap Y** to `20` in the toolbar at the top of the Room Properties form. The grid in the room will then change accordingly to make it easier to place the walls.
5. Switch to the **objects** tab. Select the wall object and place wall instances all the way along the left, top, and right boundaries of the screen. Remember that you can hold the Shift key to add multiple instances and use the right mouse button to delete them.
6. Select the Katch object and place one instance in the middle at the bottom of the screen. Also add one instance of the Pop object somewhere in the center of the room. The room should now look like Figure 6-8.

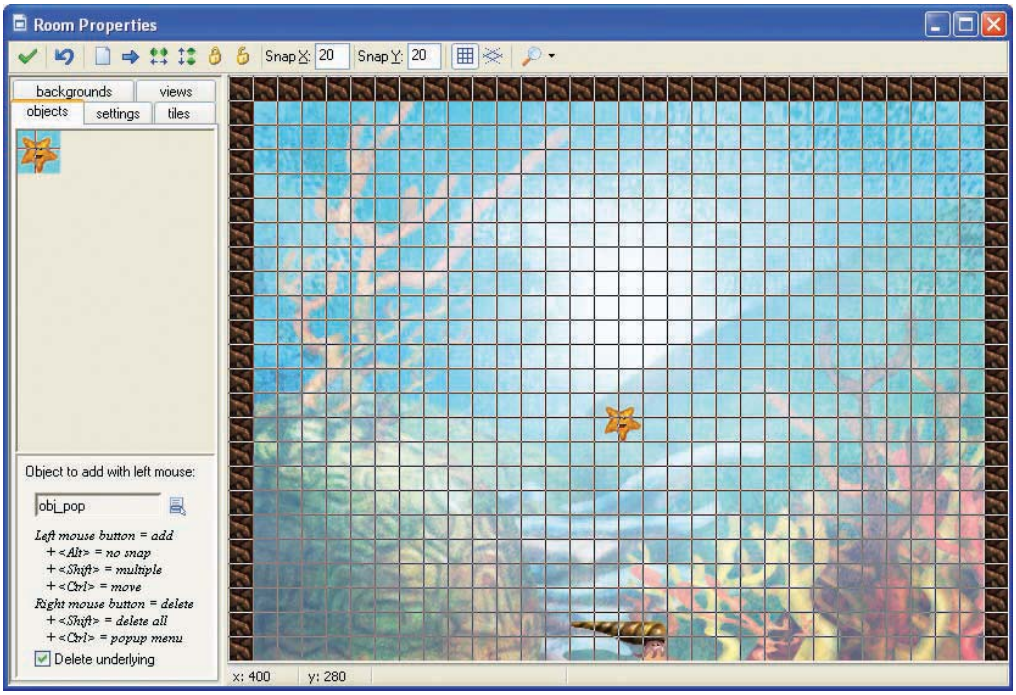

Figure 6-8. *The test room is empty, but functional.*

Note Sometimes when you close a room form you get a warning message saying that there are instances outside the room. This can happen when you accidentally move the mouse outside the room area while adding objects. You will be asked whether these instances should be removed—simply click the **Yes** button.

Now we can test our game, so save, run, and play the game. Try out the test level and make sure that Pop bounces off the walls and Katch's shell as planned. If there are problems, check your steps or load the current version from `Games/Chapter06/rainbow2.gm6` on the CD.

Biglegs

It's about time we put some goals into the game, so we're going to create objects for the Biglegs next. Pop must destroy all of the Biglegs on each level by colliding with them. The level is completed when all the Biglegs have been destroyed, so we're also going to create a *controller object* to check when all the Biglegs have gone. Despite their evil reputation, basic Biglegs will do very little and just destroy themselves when Pop collides with them.

Creating Bigleg object resources for the game:

1. Create a sprite called `spr_bigleg` using `Bigleg.gif` from `Resources/Chapter06` on the CD.
2. Create an object called `obj_bigleg` and give it the Bigleg sprite.

3. Add a **Collision** event with Pop and include a **Destroy Instance** action (**main1** tab).

4. Include a **Set Score** action (**score** tab), with **New Score** set to `200` and **Relative** enabled (remember that the **Relative** option will add the value to the current score).

Next we'll create our controller object. Each level of the game (every room except the front-end and completion rooms) will need an instance of the controller object to count the number of Bigleg instances. It will do this in every step, and when it reaches 0 it will pause briefly before advancing to the next room.

Creating a controller object resource for the game:

1. Create an object called `obj_controller`. No sprite is required.

2. Add the **Step**, **Step** event and include the **Test Instance Count** action in it (**control** tab). Indicate the Bigleg object as the object to count and leave the default values to check when the number of Biglegs is equal to zero.

3. Include the **Start Block** action to begin a block of actions that depend on the check.

4. Include a **Sleep** action (**main2** tab) with **Milliseconds** set to `1000` (1 second).

5. Include a **Next Room** action (**main1** tab).

6. Finish the current block by including an **End Block** action.

Next we'll add some instances of these objects to a couple of test rooms. We'll make two rooms by copying the first test room; this saves us from having to set the background and place the walls again.

Duplicating test room resources for the game:

1. Reopen the test room by double-clicking on it in the resource list.
2. Put one instance of the controller object into the room. It doesn't matter where, so put it somewhere like the bottom-left corner where it will be out of the way of other instances.
3. Put two instances of the Bigleg object in the top-left and -right corners of the room.
4. Close the properties form for the room.
5. Right-click on the test room in the resource list and choose **Duplicate** from the pop-up menu. This will create a copy of the room and open its properties form.
6. Switch to the **settings** tab and give the room a better name (change the caption too if you like).
7. Switch to the **objects** tab and use the right mouse button to remove the two Bigleg instances.
8. Add a few instances of the Bigleg object in the top center of the room instead. Also add some instances of the wall object below them to make it harder for Pop to reach them. This room should then look something like Figure 6-9.

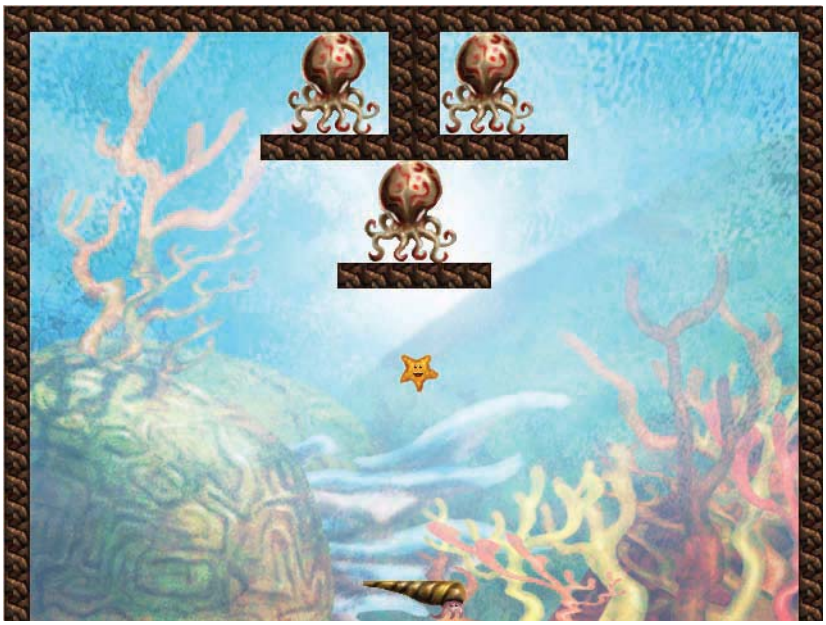

Figure 6-9. *This test room already provides a challenge.*

Save and test the game at this point to see whether it works correctly.

We now have all the ingredients we need to make a simple game, but it would be nice if there were some variation in the kinds of Biglegs that the player encounters through the game. An obvious candidate is a smaller Bigleg, making it more difficult to hit. To make this, we could repeat all the earlier steps and let the controller object count both the normal objects and the small objects. However, there is a much quicker and easier way to do this: by using *parents*. We'll explain how these work shortly, but for now follow these steps to see how powerful this feature of Game Maker can be.

Creating the small Bigleg object resource:

1. Create a sprite called `spr_bigleg_small` using the file `Bigleg_small.gif`.
2. Create an object called `obj_bigleg_small` and give it the small Bigleg sprite.
3. Click the menu icon next to the **Parent** field (on the left of the new object's properties form) and select `obj_bigleg` as the parent. The left part of the Object Properties form should now look like Figure 6-10.
4. And that's all you need to do! Close the form by clicking the **OK** button.

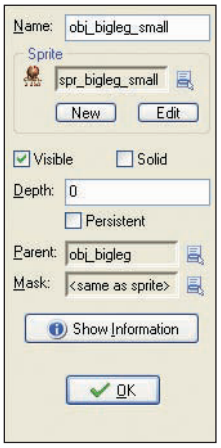

Figure 6-10. *Set the parent field for the small Bigleg.*

We now have a small Bigleg object that does everything that the normal Bigleg object does. This is because the small Bigleg inherits the behavior (all the events and actions) of the normal Bigleg when we make the normal Bigleg its parent. This means it reacts in the same way to collisions with Pop and will automatically know to increase the player's score by 200 and destroy itself. There's no need to add these events and actions again since they are automatically inherited from the parent. Also, as a child of the Bigleg, the small Bigleg is now considered to be a Bigleg too. This means that the controller object automatically includes it in its count to see how many Biglegs are left in the room.

Try this out by adding some small Biglegs to the test rooms. The small Biglegs are destroyed when Pop hits them, and the level is only completed when all the Biglegs of both types have been removed. We can use this same parenting technique to create a moving Bigleg that inherits all the same behaviors but adds a new behavior of its own as well.

Creating the moving Bigleg object resource:

1. Create an object called `obj_bigleg_move` and give it the normal Bigleg sprite.
2. Click the menu icon next to **Parent** and select `obj_bigleg` as the parent. Also set the **Depth** field to `10` so that it appears behind other objects.

3. Add a **Create** event and include the **Move Fixed** action in it. Select both the left and right directions (to make Game Maker choose randomly between them) and set **Speed** to `8`.

4. Add a **Collision** event with the wall object and include the **Reverse Horizontal** action in it. Close the form by clicking the **OK** button.

This moving Bigleg automatically inherits the behavior of the normal Bigleg (to destroy itself when Pop collides with it) and adds to that some extra behavior to make it move back and forth. In the same way, we can quickly add a small moving target.

Creating the small moving Bigleg object resource:

1. Create an object called `obj_bigleg_move_small` and give it the small Bigleg sprite.
2. Click the menu icon next to **Parent** and select `obj_bigleg_move` as the parent. Also set the **Depth** field to `10` so that it appears behind other objects.
3. Close the form by clicking the **OK** button.

This time were using the moving Bigleg as the parent, which in turn has the normal Bigleg as a parent. This means it inherits the behavior of both the moving Bigleg and the normal Bigleg. Add a few small moving Biglegs to the test levels and carefully check that the game is working correctly. You will find a version of the game so far in `Games/Chapter06/rainbow3.gm6` on the CD.

Parent Power

As you have seen, parents are extremely powerful and can save you a lot of time. The Biglegs are quite simple, but parents can save you hours of repeated work for objects with many events and actions. You could just duplicate objects to save time, but any changes you want to make afterward have to be made to both the original and each of the copies you made. However, when you change a parent, the changes automatically apply to the children of that parent too—a very useful feature.

Next we present a number of rules that determine the way that parents work in different situations. We'll describe these now for future reference, but don't worry if they don't make complete sense on your first read. Everything should become a lot clearer once you are working with real examples, so bend down the corner of the page so that you can refer back to them when you're using parents in your own games.

Inheriting Events. *A child inherits all the events (and actions) of its parent. A child can have its own events as well, but these only apply to the child and not the parent. When both the child and parent have the same event with different actions, then the actions of the child are used for the child, and the actions of the parent are used for the parent.*

We saw this earlier when the moving Biglegs inherited events from normal Biglegs but added their own extra events to make them move as well. Naturally, this did not make the normal Biglegs move as well because parents don't inherit events from children.

Actions on Objects. *When an action refers to a parent object, this includes instances of the child object as well. However, when an action refers to the child object, it does not include instances of the parent object.*

So when counting the number of instances of normal Biglegs, it automatically included all of the other Bigleg objects that were children of it. The same rule applies to any action that can be applied to an object. However, remember that **Self** and **Other** refer to instances, not objects, so applying an action to them does not affect their parent or child objects.

Collision with Objects. *A collision event with a parent object also applies to collisions with children of that object. However a collision event with a child object does not apply to collisions with parents of that object.*

This will be useful later on as we want to have many different types of coral blocks that Pop can collide with. By making all of these blocks have the same parent, we only need to define one collision event between Pop and the parent block, and it will also work for all the child blocks.

Parenting Objects. *Parents can have parents, which can have parents, etc. However, you must not create cycles of parents, so if P is the parent of C, then C cannot also be the parent of P.*

This may sound confusing, but makes perfect sense if you think about it—you couldn't be your own father or grandfather, so it doesn't work in Game Maker either.

Lives

In this kind of game, it is common to give the player a fixed number of lives to try and complete it. Game Maker includes events and actions specifically to handle lives. These allow you to set and display the number of lives as well as testing for when the player has none left. We'll use our controller object to look after the actions that control lives, but first we need to set the number of lives at the start of the game. The best place to do this is in the title object.

Setting the lives in the create event of the title object:

1. Double-click the title object in the resource list to open its properties form.

2. Select the **Create** event to view its **Actions** list. Include a **Set Lives** action (**score** tab), with a value of `3`.

Next we'll make the player lose a life when Pop falls out of the room. However, one of our planned features for the game will make copies of Pop so that there can be several Pops flying around the screen at the same time. So the player should only lose a life when the last surviving Pop falls out of the room. When this happens, we'll reduce the player's lives and create a new Pop in the center of the screen.

Editing the Pop object to add an event for being outside of the room:

1. Double-click the Pop object in the resource list to open its properties form.

2. Select the **Outside Room** event and remove the **Restart Room** action from it (left-click on the action once and press the delete key, or right-click on the action and select delete). Include a **Destroy Instance** action in its place.

3. Include the **Test Instance Count** action (**control** tab). Indicate the Pop object as the object to count and leave the default values to check if this is the last Pop leaving the screen.

4. Include the **Start Block** action to begin a block of actions that depend on the test.

5. Include a **Set Lives** action with **New Lives** set to `-1` and the **Relative** option enabled. This will then subtract one from the number of lives.

6. Include a **Create Instance** action. Select the Pop object; set **X** to `320` and **Y** to `300`.

7. Finally, include an **End Block** action and close the Pop object's properties.

When the player loses their last life, we need to show the high-score table and return to the front-end room. We'll use the controller object to check this using a special event.

Editing the controller object to add an event for having no more lives:

1. Double-click the controller object in the resource list to open its properties form.

2. Add the **Other, No More Lives** event and include a **Show Highscore** action in it. Use the same **Background** and **Font** as before.

3. Include the **Different Room** action and indicate the front-end room. Close the controller object's properties form.

Finally, we should display the number of lives on the screen—otherwise, the end of the game could be a bit of a shock to the player! To do this, we're going to use a new event called the **Draw** event, which requires a little explanation. Normally each object automatically draws its own sprite in the correct position on the screen. However, the **Draw** event allows you to draw something else instead, such as several different sprites or colored shapes and text. When the **Draw** event is included, the object's sprite stops being drawn automatically and your own actions in the **Draw** event are executed instead.

We're going to use the **Draw** event of the controller object to show the player's remaining lives as a number of small shells along the bottom of the screen. Game Maker even provides the **Draw Life Images** action to help us do this.

Creating a sprite for the controller object to draw as lives:

1. Create a sprite called `spr_katch_small` using the file `Katch_small.gif`.
2. Double-click on the controller object in the resource list to open its properties form.
3. Add the **Draw** event and include the **Draw Life Images** action (**score** tab). Set **X** to `25` and **Y** to `470`, and select the small Katch sprite (as shown in Figure 6-11).

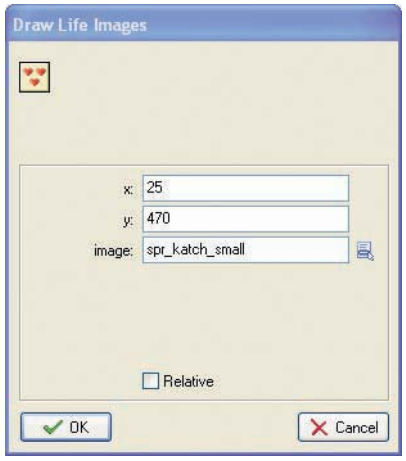

Figure 6-11. *This action will draw the lives as images.*

Note Adding a **Draw** event to any object stops that object from automatically drawing its own sprite and allows you to include your own draw actions instead. This means that if you want to draw something *in addition* to the object's normal sprite, you need to draw the normal sprite as well. If you always know what sprite this should be, then simply use a **Draw Sprite** action. However, if you want to be able to change the sprite using the **Set Sprite** action, then include an **Execute Code** action (**control** tab) and type `draw_sprite(sprite_index, image_index, x, y)` into it. This small piece of GML code (see Chapter 12) will then draw the sprite in the usual way *in addition* to any other draw actions in the event.

When you play the game now, there should be three small images of Katch in the bottom left of the screen. One should disappear each time Pop goes off the screen, and the game ends with a high-score table when they're all gone. You'll find a version of the game so far in `Games/Chapter06/rainbow4.gm6` on the CD.

Blocks

There are only so many ways of varying a level by placing Biglegs in different places, so we're going to provide them with some defenses to give the level design a bit more scope. We'll start with simple (coral) blocks that deflect Pop but are destroyed in the process. These blocks are just there to get in the way, but later we'll go on to make some special blocks as well.

Normal Blocks

We can use parents to quickly create a number of different-colored blocks. These blocks will all behave in exactly the same way, and the different colors are just to make the Rainbow Reef live up to its name.

Creating block object resources for the game:

1. Create seven sprites called `spr_block1` to `spr_block7` using the files `Block1.gif` to `Block7.gif`. Disable the **Transparent** option on them all to make them appear solid.
2. Create a new object called `obj_block1` and give it the first block sprite. Enable the **Solid** option and close the properties form. That's it for this object.
3. Now create six more objects for the other six blocks using the appropriate sprites. Enable the **Solid** option on each one and set **Parent** to `obj_block1`. They are now all children of the first block.

Editing the Pop object to add a collision event with blocks:

1. Reopen the Pop object's properties form from the resource list.

2. Add a **Collision** event with `obj_block1` and include the **Bounce** action with **Precise** set to **precisely**.

3. Include the **Set Score** action with **New Score** set to `20` and the **Relative** option enabled.

4. Finally, include a **Destroy Instance** action and set **Applies to** to **Other**. This indicates that it is the block rather than Pop that should be destroyed.

This is the only collision event that we need to define between Pop and the seven types of block. This is because all the other blocks are children of the first block, so this event applies to collisions with those blocks as well. Add some of the new blocks to the test levels and play the game to make sure everything works as expected.

Solid Blocks

Next we'll create a solid block that Pop can't destroy. This behaves in the same way as the wall object, so we'll use that as its parent.

Creating a solid block resource for the game:

1. Create a sprite called `spr_block_solid` using `Block_solid1.gif`. Disable the **Transparent** option to make it appear completely solid.
2. Create a new object called `obj_block_solid` and give it the solid block as a sprite. Check the **Solid** option and give it the wall object as a **Parent**.

We do not need to define any collision events between Pop and solid blocks because we've already defined collision events with its parent, the wall. Let's also create an invisible solid block to give the player a surprise in later levels. This is actually probably not such a good idea from a design perspective. Players will just think it's a bug or feel unfairly punished when Pop bounces off in an unexpected direction. However, it illustrates a point about collisions with invisible objects, so we'll make it as an example.

Creating an invisible block resource for the game:

1. Create a new object called `obj_block_solid_inv` and give it the solid block as a sprite. Enable the **Solid** option and give it the wall object as a **Parent**. Also disable the **Visible** option to make the object invisible.

So even though this object is invisible, it still needs a sprite. Game Maker needs a sprite to work out when objects collide with each other, and an object without a sprite will not trigger any collision events.

Special Blocks

Now let's add a few more interesting blocks. This one will require two hits to destroy it.

Creating the double block sprite and object resources:

1. Create a sprite called `spr_block_double` using `Block_double.gif`. Disable the **Transparent** option to make it appear completely solid.
2. Create a new object called `obj_block_double`. Give it the double sprite and enable the **Solid** option. Close the object properties.

Adding a collision event to the Pop object for colliding with the double block:

1. Reopen the Pop object's properties form and add a **Collision** event with the double block object.

2. Include the **Bounce** action in this event and set **Precise** to **precisely**.

3. Also include a **Set Score** action with a **New Score** of `20` and the **Relative** option enabled.

4. Finally, include a **Change Instance** action. Set **Applies to** to **Other** and select `obj_block1` so that the double block changes into a normal block.
5. Close the Object Properties form.

Next we'll add a block that creates two additional instances of Pop, making it look as if he has split into three (like some starfish can do).

Creating the split block sprite and object resources:

1. Create a sprite called `spr_block_split` using `Block_split.gif`. Disable the **Transparent** option to make it appear completely solid.
2. Create a new object called `obj_block_split` and give it this new sprite. Enable the **Solid** option and set the **Parent** field to `obj_block1`.
3. Add a **Collision** event with the Pop object and include a **Create Instance** action. Set **Object** to `obj_pop`, then type `other.x` into **X** and `other.y` into **Y** (see Figure 6-12). This will create the new Pop at the same position as the old Pop.
4. Include another **Create Instance** action exactly the same as in step 3.

Figure 6-12. *Let's create some extra starfish.*

Finally, let's add a block that gives the player an extra life.

Creating the life block sprite and object resources:

1. Create a sprite called `spr_block_life` using `Block_life.gif`. Disable the **Transparent** option to make it appear completely solid.
2. Create a new object called `obj_block_life`. Enable the **Solid** option and set the **Parent** field to `obj_block1`.
3. Add a **Collision** event with the Pop object and include a **Set Lives** action. Set **New Lives** to `1` and enable the **Relative** option.

It is time to test all these blocks by adding them to our test levels. Feel free to experiment and see what incredible combinations of levels you can come up with, or just use ours by loading the file `Games/Chapter06/rainbow5.gm6` from the CD.

Polishing the Game

We've already done most of the hard work, so now it's time to put everything together into a playable game.

Sound Effects

All the games we've made so far have benefited from sound effects, and this is no exception. As you saw in the previous chapter, sounds help the player to correctly interpret their interactions with the game. We'll add different sounds for when Pop hits a wall, a block, a Bigleg or Katch's shell, as well as when Pop falls out of the screen.

Creating sound resources and playing them in the appropriate events:

1. Create sounds using the files `Sound_wall.wav`, `Sound_block.wav`, `Sound_katch.wav`, `Sound_bigleg.wav`, `Sound_lost.wav`, and `Sound_click.wav`, and give them appropriate names.

2. Play the "wall sound" in the Pop object's **Collision** event with the wall object.

3. Play the "block sound" in the Pop object's **Collision** event with the block object and the double block object.

4. Play the "katch sound" in the Pop object's **Collision** event with the Katch object.

5. Play the "lost sound" in the Pop object's **Outside Room** event.

6. Play the "bigleg sound" in the Bigleg object's **Collision** event with the Pop object.

7. Play the "click sound" at the top of the **Left Pressed** event of each of the button objects.

Saving Games and Quitting

If you can remember back to when we were making the framework, we included a button in the front-end to load a saved game. However, in order for this to work, we need to allow the player to save the game at some point using the **Save Game** action. We'll add an event for this on the controller object when the S key is pressed.

■ **Note** Game Maker must be able to write to the filename you provide in the **Save Game** action. This wouldn't normally be a problem, but it does mean you'll get an error message if you run many of the example games directly from the CD-ROM. This is because Game Maker cannot save files to a CD, so copy the game files to your hard disk and run them from there in order to fix the problem.

Adding a save game action to the controller object:

1. Reopen the controller object's properties form from the resource list.

2. Add a **Key Press, Letters, S** event and include the **Save Game** action (**main2** tab). You can leave **File Name** set to the default as it is the same as we used for the **Load Game** action in the front-end.

3. Include the **Display Message** action (**main2** tab) and type "GAME SAVED" into **Message**, so that the user knows the game was saved at this point.

4. Add a line to the **Game Information** describing how the game can be saved.

When you press the Esc key during play, the program ends completely, but it would be better if this returned the player back to the front-end. Adding actions to return to the front-end is simple enough, but this won't work on its own. We also need to disable the default behavior of the Esc key from within the **Global Game Settings**.

Editing the controller object and global game settings to disable the Esc key:

1. Reopen the controller object's properties form from the resource list.

2. Add a **Key Press, Others, <Escape>** event and include the **Different Room** action (**main1** tab). Select the front-end room and close the controller object's properties form.

3. Double-click the **Global Game Settings** at the bottom of the resource list.

4. Switch to the **Other** tab and disable the **Let <Esc> end the game** option.

5. Also disable the **Let <F5> save the game and <F6> load a game** option as we have our own save and load system.

6. Click **OK** to close the form.

There are many other settings that can be changed in the global game settings, but most of these are for advanced use. We will learn more about some of them later in the book.

Caution Beware of disabling the **Let <Esc> end the game** option and forgetting to include your own method for quitting your game (using the **End the Game** action). If this happens, you'll need to press Ctrl+Alt+Delete to bring up the Task Manager and end the task from the Applications list. This will then end the game and return you to Game Maker.

A Slower Start

Currently, Pop starts moving as soon as the level begins, giving the player no time at all to find their bearings. The player may have been at the far side of the screen when Pop last fell out, so it would be fairer to give them a little time to move Katch into a better position. To avoid any

complications when new Pop instances are created during play, we're going to make a second, stationary Pop object. This will have an **Alarm Clock** in its **Create** event that changes it into the normal Pop when the time is up. This stationary Pop object will then be the one that gets positioned in each room at the start of the game.

Creating a new stationary Pop object resource:

1. Create a new object called `obj_pop_start`. Give it the Pop sprite and a **Depth** of `10`.

2. Add a **Create** event and include a **Set Alarm** action (**main2** tab) with `30` **Steps** (1 second).

3. Add an **Alarm, Alarm 0** event and include the **Change Instance** action in it. Set **Change Into** to `obj_pop` and indicate **yes** to **Perform Events** (see Figure 6-13). This will ensure that the new Pop object starts moving in a random direction by performing its **Create** event.

4. Reopen the Pop object's properties form and select the **Outside Room** event. Double-click the **Create Instance** action and change the Pop object to `obj_pop_start`.

5. Go through each room replacing the Pop objects with the start Pop object.

Figure 6-13. *This action changes the stationary Pop into a normal Pop and performs the* ***Create*** *event.*

Creating the Levels

Before we start creating the final levels, we'll add some "cheats" to make it possible to skip through the levels for testing them. We'll use the N key to move to the next level, the P key to go to the previous level, and R key to restart the current level. We'll also add a cheat that adds an extra life when you press the L key.

Editing the controller object to add cheats:

1. Reopen the controller object's properties form from the resource list.

2. Add a **Key Press, Letters, N** event and include the **Next Room** action.

3. Add a **Key Press, Letters, P** event and include the **Previous Room** action.

4. Add a **Key Press, Letters, R** event and include the **Restart Room** action.

5. Finally, add a **Key Press, Letter, L** event and include the **Set Lives** action. Set **New Lives** to `1` and enable the **Relative** option.

Now it's finally time to create your levels. Let your imagination run wild and see what you can come up with. It will save you time to begin by making a standard room containing only the walls, the controller, the stationary Pop, and Katch. You can then copy this each time and add the Biglegs and blocks for that level. Good level design is crucial for the success of the game and more difficult than you may think. However, you'll learn a lot by just trying it for yourself, and we'll discuss it further in Chapter 8.

Congratulations

Phew—and that's another one done! You'll find the final version of the game in the file `Games/Chapter06/rainbow6.gm6` on the CD. We've designed eight different levels—why not play them and see if they give you any ideas for your own? There's plenty of scope for expanding the features of the game, if you want to. Here are some suggestions:

- Add other types of Biglegs that move in different directions.
- Add blocks that move around.
- Add bonus blocks that give a large number of points to the player's score.
- Add some bad blocks that decrease the score! (Be sure that you make it obvious to the player that this is happening—perhaps by using a new sound effect.)
- Add blocks that increase or decrease the speed of Pop.
- Add blocks that change the size of Pop or Katch.

In this chapter, you have learned about parents and used them in a game for the first time. Parents are very powerful and can save you a lot of work when you are making and changing your games. We strongly encourage you to make sure that you understand them so that you can make full use of them in your own games.

We also introduced our game framework, which we'll use again and refer back to in later chapters. You saw how to deal with lives and how to use draw events and drawing actions. Finally, you used new events and actions introducing gravity, bouncing, and alarm events into your games.

You've already become quite experienced at Game Maker and made some enjoyable games. However, we have a real treat for you in next chapter as we join a cute band of koala bears who just can't help walking into dangerous objects . . .

CHAPTER 7

Maze Games: More Cute Things in Peril

Maze games have been popular since the days of Pac-Man, and they're another kind of game that's easy to make in Game Maker. In this chapter, we'll create a puzzle game where the player must help koala bears escape from a maze full of hazardous obstacles. The focus of the game will be on puzzles rather than action, and the levels will be designed to make the player think carefully about the strategies they must use to avoid any unpleasant accidents. (No animals were hurt in the making of this game.)

Designing the Game: Koalabr8

The name *Koalabr8* is a play on the word "Collaborate" with a bit of text-speak thrown in for good measure. This is because the puzzle of the game is based on the idea of controlling many koalas at the same time. So, for example, when you press the up key, all the koalas in your team will move up together. If you imagine trying to steer several koalas through a minefield in this way, then you'll get a sense of the kind of challenge we're aiming for (see Figure 7-1). Anyway, here's the full description of the game:

A colony of koala bears have been captured by the evil Dr. Bruce for use in his abominable experiments. The koalas manage to escape from their cages only to find that the doctor has implanted some kind of mind control device in their brains. The only way they can overpower the controlling effect is to combine their thoughts and all perform the same actions at once. The koalas must work together to find their way past the many dangers in the doctor's laboratory and escape to freedom.

The arrow keys will simultaneously move all of the bears on a level, except bears whose paths are blocked by a wall or another bear. Each level will be a hazardous maze that is completed by getting all of the koalas to an exit. However, if a koala touches a dangerous hazard on the way, then he dies and the level must be replayed. The game will contain a number of fatal and nonfatal hazards shown in the following feature list:

- *Fatal hazards*
    - *Explosive TNT*
    - *Moving circular saws*

- *Nonfatal hazards*
  - *Red exits—Allow any number of koalas to exit the level*
  - *Blue exits—Allow a single koala to exit the level*
  - *Locks—Block the path of koalas (red, blue, and green)*
  - *Switches—Open locked passageways (normal, timed, and pressure)*
  - *Boulders—Can be pushed by koalas and destroy other hazards*

Figure 7-1. *Here's a typical level in the Koalabr8 game.*

As always, you'll find all the resources for this game in the `Resources/Chapter07` folder on the CD.

The Basic Maze

We'll start by making a basic maze and getting a koala to walk around it. This same technique can be used for making any kind of maze game, so feel free to copy it for your own projects.

The Game Framework

This game will use the same basic framework as we made in the previous chapter. This consists of a front-end with buttons to start a new game, load a saved game, show help, and quit the game, as well as a completion screen that congratulates the player. However, there is no score in this game so there will be no high-score table. The instructions that follow will show you how to create the front-end for this game, although you might want to refer back to Chapter 6 for a more detailed explanation. Alternatively, you can just load the completed framework from `Games/Chapter07/koala1.gm6` on the CD and skip to the next section, "A Moving Character."

Creating the front-end:

1. Launch Game Maker and start a new game from the **File** menu.
2. Create sprites using the following files from the `Resources/Chapter07` folder on the CD: `Title.gif`, `Button_start.gif`, `Button_load.gif`, `Button_help.gif`, and `Button_quit.gif`. Remember to name them appropriately.
3. Create a background using the file `Background.bmp`.
4. Create sounds using the files `Music.mp3` and `Click.wav`.

5. Create a title object using the title sprite. Add an **Other, Game Start** event and include a **Play Sound** action, with **Loop** set to true.

6. Create a start button object using the start sprite. Add a **Mouse, Left Pressed** mouse event and include an action to play the click sound followed by an action to move to the next room.

7. Create a load button object using the load sprite. Add a **Mouse, Left Pressed** mouse event and include an action to play the click sound followed by an action to load the game (use the default file name).

8. Create a help button object using the help sprite. Add a **Mouse, Left Pressed** mouse event and include an action to play the click sound followed by an action to show the game information.

9. Create a quit button object using the quit sprite. Add a **Mouse, Left Pressed** mouse event and include an action to play the click sound followed by an action to end the game.
10. Create a room using the background, and place the title and four button objects in it so that it looks like Figure 7-2.

Figure 7-2. *The finished front-end for Koalabr8 should look something like this.*

Now follow these instructions to create the completion screen. Refer to Chapter 6 for a more detailed explanation.

Creating the completion screen:

1. Create a sprite using the file `Congratulation.gif`.
2. Create a new object using this sprite. Add a **Create** event and include a **Set Alarm** action to set **Alarm 0** using `120` **Steps**.
3. Add an **Alarm, Alarm 0** event and include an action to move to the front-end room.
4. Create a completion room using the background and place an instance of the congratulations object in it.

We also need to create the game information and change some of Game Maker's default settings for the game.

Changing the game settings:

1. Double-click **Game Information** near the bottom of the resource list and create a short help text based on the game's description.
2. Double-click **Global Game Settings** at the bottom of the resource list.
3. Switch to the **Other** tab and disable the two options **Let <Esc> end the game** and **Let <F5> save the game and <F6> load a game** as we handle this ourselves.

That completes our game framework for Koalabr8, which can also be loaded from the file `Games/Chapter07/koala1.gm6` on the CD.

■ **Caution** Beware of disabling the **Let <Esc> end the game** option and forgetting to include your own method for quitting your game (using the **End the Game** action). If this happens, you'll need to press Ctrl+Alt+Delete to bring up the Task Manager and end the task from the Applications list. This will end the game and return you to Game Maker.

A Moving Character

The basis of every maze game is a character that moves around walled corridors, so we will begin by creating a wall object.

Creating the wall object:

1. Create a sprite called `spr_wall` using the file `Wall.gif`. Disable the **Transparent** option, as our wall is just one large solid block.
2. Create a new object called `obj_wall` using this sprite and enable the **Solid** option.

You might be wondering why we are using an ugly black square for our walls. Don't worry; we'll transform it into something that looks much nicer when we learn how to use *tiles* later on in the chapter.

Next we need to create a character, which in this case is a koala bear. We'll use five different sprites for the koala: four animations for walking in each of the four directions and another sprite for when koalas are standing still.

Creating the koala sprites:

1. Create a sprite called `spr_koala_left` using the file `koala_left.gif` and enable the **Smooth edges** option.
2. Create sprites in the same way using the files `koala_right.gif`, `koala_up.gif`, `koala_down.gif`, and `koala_stand.gif`.

Now we can create our koala object and give it actions that allow the player to move it around the screen. You can probably work out how to do this yourself by now, using either **Jump** or **Move** actions in a similar way to one of the previous games. You may even like to have a try for yourself before continuing, but you'll actually find that it's very tricky to move a character through a maze using just these actions. This is because the sprites of both the walls and koalas are exactly 40×40 pixels, so all the corridors are only just big enough for the koalas to walk down. You'll see the problem if you imagine running through a maze with your arms fully stretched out, where the width of the corridors is exactly the same as your arm span! Bumping into walls and struggling to change direction in corridors soon removes all feeling of control and the game stops being fun.

Fortunately, Game Maker comes to the rescue with the **Check Grid** action. This conditional action allows us to test for when a koala is exactly lined up with the corridors. This means we can ignore the player's badly timed key presses (that normally cause the koala to walk into walls) and wait until Game Maker knows the koala is in exactly the right place. Only then do we let the player stop or turn their character, so that the koalas end up gliding gracefully along the corridors with no fuss at all.

In addition to checking that koalas are aligned correctly, we will check that there is nothing blocking their way before even starting to move. This results in a reliable control system that feels slick to the player and is a solid basis for any kind of maze game you might be making. Follow these steps and you'll see what we mean.

Creating the koala object:

1. Create a new object called `obj_koala` using the standing koala sprite. Set the object's **Parent** to be the wall object—this may sound odd, but all will be explained in the following steps.

2. Add a **Keyboard, <Left>** event and include the **Check Grid** action (**control** tab). Set both **Snap hor** (horizontal snap) and **Snap vert** (vertical snap) to `40` to indicate a grid size of 40×40 pixels. Also enable the **NOT** option so that the event checks for koalas not being aligned with the grid. The form should look like Figure 7-3.

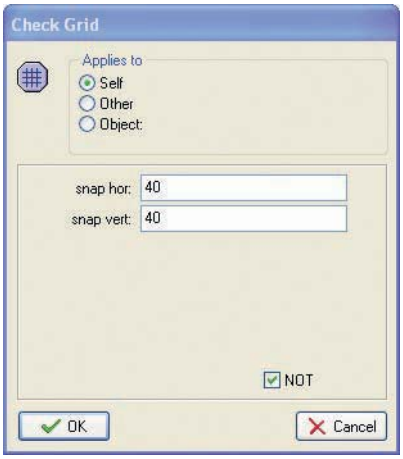

Figure 7-3. *Check whether the instance is not aligned with the grid.*

3. Include the **Exit Event** action. This action stops any more actions from being performed in the **Actions** list. We have put it underneath our **Check Grid** condition so that none of the following actions are performed when the koala is *not* aligned with the grid.

4. Include the **Check Object** action. Indicate the wall object and enable the **NOT** option. Set **X** to `-40` and **Y** to `0` and then enable the **Relative** option so that it checks that there are no wall objects one grid square to the left of the koala.

5. Include the **Start Block** action followed by the **Move Fixed** action. Select the left arrow and set **Speed** to `5` (this speed must divide exactly into the grid size of 40; otherwise the koalas will not stop on each grid square).

6. Include the **Change Sprite** action and indicate the left-facing koala sprite. Set **Subimage** to `-1` so that the animation keeps playing despite the sprite being changed.

7. Those are all the actions we need to move the koala, so include an **End Block** action.

8. However, if there is a wall in the way, then we must stop the koala from moving. We've already checked for walls *not* being present using the **Check Object** action, so including an **Else** action will allow us to define what should happen when a wall *is* present. Include the **Else** followed by a new **Start Block** action.

9. Include the **Move Fixed** action, select the middle square, and set **Speed** to `0`.

10. Include the **Change Sprite** action using the standing sprite with a **Subimage** of `-1`.

11. Finally, include the **End Block** action. The actions should now look like Figure 7-4.

12. Now repeat steps 2–11 to create similar rules for the Right, Up, and Down arrow keys. Make sure you choose the correct direction for each **Move Fixed** action, the correct sprite for each **Change Sprite** action, and the correct **X** and **Y** values for each **Check Object** action. When you're making the up and down rules, remember that y increases as you move down the screen.

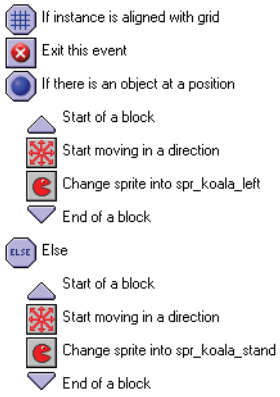

Figure 7-4. *These are the actions for moving the koala to the left. Note that the first condition has its **NOT** property set, so it checks for when the instance is not aligned with the grid.*

Before continuing, check through your actions and make sure that you can logically follow what they do.

The **Keyboard** actions start the bear moving if there isn't a wall in the way, but we haven't added any actions to stop the bear yet. This will happen automatically if the player keeps their finger on one of the arrow keys, as the actions in the **Keyboard** events will eventually detect a wall and stop. However, the bear also needs to come to a halt if the player stops pressing any

arrow keys. If you're thinking that we need a **<No Key>** event, then you're on the right track, but if we stopped bears moving in a **<No Key>** event, then what would happen if the player was pressing another key such as the spacebar? The spacebar isn't used in this game, but it's still a key—so the **<No Key>** event would never be called and the koala would never stop!

Instead of using a **<No Key>** event, we're going to use a **Begin Step** event. This will keep checking if the bear has reached the next grid square, and stop it when it does. We need to use a **Begin Step** rather than any other kind of **Step** event, because the **Begin** part means that the actions will be called at the beginning of each step—before the **Keyboard** events. Therefore, we can stop the bear moving in **Begin Step** and start it again in **<Left>** if the player is pressing the Left key. If we used a **Step** or **End Step**, then these events would happen the other way around: **<Left>** would start the bear moving and **Step** would stop it again—canceling the effect of the pressing the key. Once you've added the actions that follow, try changing the **Begin Step** event to a **<No Key>** or a normal **Step** event and seeing what effect it has on the game.

Adding a Begin Step event to the koala object:

1. Add a **Step, Begin Step** event to the koala object and include the **Check Grid** action. Set **Snap hor** to 40 and **Snap vert** to 40, but this time leave **NOT** disabled.

2. Include a **Start Block** action followed by a **Move Fixed** action. Set **Speed** to 0 and select the center square to stop the koala moving.

3. Include a **Change Sprite** action, indicating the standing koala sprite with **Subimage** set to -1.

4. Include an **End Block** action and close the koala object's properties.

Caution When setting a **Move Fixed** action with a speed of 0, you must also select the center square of the direction grid. If no direction square is selected at all, then the action is ignored!

Okay, hopefully that makes sense so far, but we still haven't explained why we made the wall object a parent of the koala. While this may be against the laws of nature, it's not against the laws of Game Maker, and it's actually saved us a lot of work. Koalas need to stop for both walls and other koalas so we *could* add extra actions to check for koalas in the same way we did earlier. However, this happens automatically when we make walls a parent of koalas, as Game Maker now treats koalas as a special kind of wall!

Before we can test our work, there is something we need to fix. When we were making the front-end we indicated that pressing the Esc key should not automatically end the game. However, we need *some* way of quitting a level once it is running, so we'll use a controller object to take us back to the front-end when the Esc key is pressed.

Creating the controller object:

1. Create a new object called `obj_controller` and leave the sprite as **<no sprite>**.

2. Add a **Key Press, Others, <Escape>** key event. Include the **Different Room** action and indicate the front-end room.

To test our basic maze game system, we'll need to create a test level by adding a new room between the front-end and closing rooms.

Creating a test room:

1. Right-click the completion room in the resource list and select **Insert Room** from the pop-up menu. This will insert the new room between the two existing rooms.
2. Switch to the **settings** tab and give the room an appropriate name and caption.
3. Switch to the **backgrounds** tab and give the room the background.
4. Set the **Snap X** and **Snap Y** on the toolbar to the correct cell size of 40 pixels.
5. Select the **objects** tab and create a maze out of instances of the wall object. Leave the top row free, as we'll need this space for displaying other game information later on. Place three or four instances of the koala object in the maze and put one instance of the controller object in the top-left corner of the room. The room should then look something like Figure 7-5.

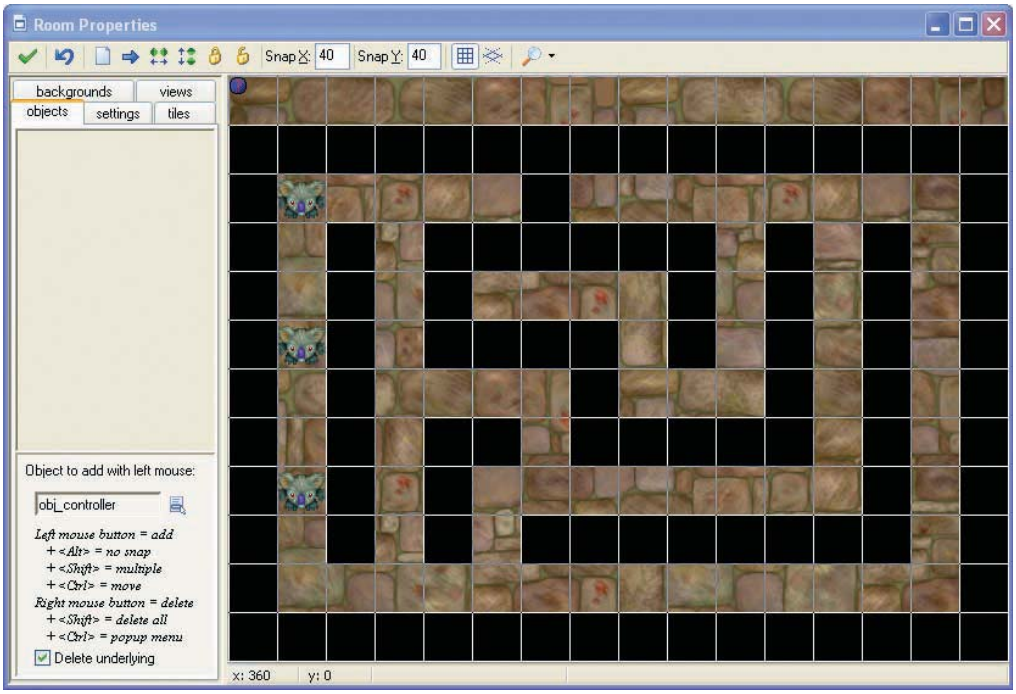

Figure 7-5. *Here's our first maze.*

Save your work and run the game to test it. Check that the koalas don't get stuck walking around and that the animations play correctly. Also notice how the koalas react when they bump into each another. If you need it, you'll find this version of the game in the file `Games/Chapter07/koala2.gm6` on the CD.

Save the Koala

Most of what we have done so far could be used for any kind of maze game, but now we'll begin adding features specific to the Koalabr8 game design. The aim of each level is to get all the koalas out alive, so in this section we'll add exits and check whether all koalas have been rescued before moving on to the next level.

We'll begin by creating objects for the two different kinds of exits: one that can be used by any number of koalas to escape, and one that is destroyed after being used by just a single koala. When a koala collides with an exit, we know it has escaped the level and so we can destroy the koala instance. We'll use Game Maker's built-in mechanism for controlling the player's lives to indicate how many koalas have been saved in each level.

Creating the exit objects:

1. Create sprites called `spr_exit1` and `spr_exit2` using `Exit1.gif` and `Exit2.gif` and enable the **Smooth edges** option for them both.

2. Create an object called `obj_exit1`, and give it the first exit sprite and a **Depth** of `10`. We only want the koala to disappear when it's completely on top of the exit, so add a **Collision** event with the koala object and include a **Check Grid** conditional action. Select **Other** from **Applies to** (the koala) and set **Snap hor** and **Snap vert** to `40`.

3. Include a **Start Block** action.

4. Include a **Destroy Instance** action and select **Other** from **Applies to** (the koala).

5. Include a **Set Lives** action with **New Lives** set to `1` and the **Relative** option enabled.

6. Finally, include an **End Block** action and close the properties form.

7. Now create another exit object in exactly the same way using the sprite for exit2. The only difference is that you need to include an additional **Destroy Instance** action after step 4 with **Applies to** set to **Self** (the exit).

We'll use the controller object to check when all koalas have been rescued. It will also display the rescued koalas along the top of the screen using the **Draw Lives** action.

Adding events to handle rescued koalas in the controller object:

1. Create a sprite called `spr_rescued` using `Rescued.gif` and enable the **Smooth edges** option.

2. Reopen the properties form for the controller object by double-clicking it in the resource list.

3. Add a **Create** event and include the **Set Lives** action with **New Lives** set to `0`. We're using lives to represent rescued koalas—so the lives need to start at 0.

4. Add a new **Step, Step** event and include the **Test Instance Count** action. Set **Object** to the koala object and **Number** to `0`. When there are no koala instances left, then the player must have completed the level.

5. Include the **Start Block** action followed by the **Sleep** action. Set **Milliseconds** to `2000` (2 seconds).

6. Include the **Next Room** action and set **Transition** to **Create from center** (or any other one that takes your fancy).

7. Include the **End Block** action to complete the actions for this event.

8. Add a new **Draw** event. Switch to the **draw** tab and include the **Draw Sprite** action. Set **Sprite** to the rescued sprite, **X** to `10`, and **Y** to `0`. Leave the **Relative** option disabled, as we want this sprite to be drawn in the top left of the screen.

9. Include the **Draw Life Images** action with **Image** set to the standing koala sprite, **X** to `150`, and **Y** to `0`. This will draw the sprite once for each of the player's lives (saved koalas).

■ **Caution** **Draw** actions all have a light yellow background and can only be used in **Draw** events. If you put them in a different event, then they are ignored.

Now add some exits to your test room and give it a quick play to make sure they are working correctly. You might also want to add a second test room (remember that you can duplicate rooms). You'll find the current version in `Games/Chapter07/koala3.gm6` on the CD.

Creating Hazards

Our maze is rather dull at the moment, so it's time to create some challenges by making it a lot more dangerous for the koalas. We'll start by adding TNT that blows koalas off the level if they touch it and restarts the level. This may sound like an easy hazard to avoid, but carefully positioned TNT can present quite a challenge when you're controlling several koalas at once!

We'll begin by creating a dead koala (how often do you get to say that?). Dead koalas will fly off the screen in an amusing fashion—nothing too gory, as we want to keep this a family game. We'll do this using Game Maker's gravity action, as that will do most of the work for us and make the movement look realistic.

Creating the dead koala object:

1. Create a sprite called `spr_koala_dead` using `Koala_dead.gif` and enable **Smooth edges**.
2. Create a new object called `obj_koala_dead` using this sprite. Give it a **Depth** of `-10` to make sure it appears in front of all other objects.

3. Add a **Create** event and include the **Move Free** action. Set **Direction** to `80` and **Speed** to `15`.

4. Include the **Set Gravity** action with **Direction** set to `270` (downward) and **Gravity** set to `2`.

5. Add an **Other, Outside room** event and include the **Sleep** action. Set **Milliseconds** to `1000`.

6. Include the **Restart Room** action with **Transition** left as **<no effect>**. It would soon annoy the player to have to wait for a transition to restart the level each time.

Next we'll create our TNT object. It is a very simple object that just sits there and turns koalas into dead koalas when they collide with it.

Creating the TNT object:

1. Create a sprite called `spr_TNT` using `TNT.gif` and enable **Smooth edges**.
2. Create a new object called `obj_TNT` using this sprite.

3. Add a **Collision** event with the koala object. Include the **Change Instance** action and select **Other** from **Applies to**, in order to change the koala. Set **Change into** to the dead koala object and set **Perform events** to **yes**.

We've now actually created a bug (an unintentional error) in our game. Remember that the controller object moves to the next room when there are no koalas left on the level—assuming that the player must have rescued them all. However, if the last koala is killed by TNT, then there will also be no koalas left on the level—but the player has failed! Consequently, we must alter the controller object so that it also checks that there are no dead koalas before moving to the next level.

Editing the controller object to fix the dead koala bug:

1. Double-click the controller object in the resource list and select the **Step** event.

2. At the top of the action list, include another **Test Instance Count** action and set **Object** to the dead koala. As this appears directly above the old check for koalas, the block of actions will only be performed if both conditions are true.

Test this out by adding some TNT to your levels. It's actually possible to build very difficult levels just using TNT. The level, shown in Figure 7-6, looks almost impossible to solve, but it is solvable once you work out how to use the extra wall piece on the left. It's designing puzzles like this that will make this game interesting.

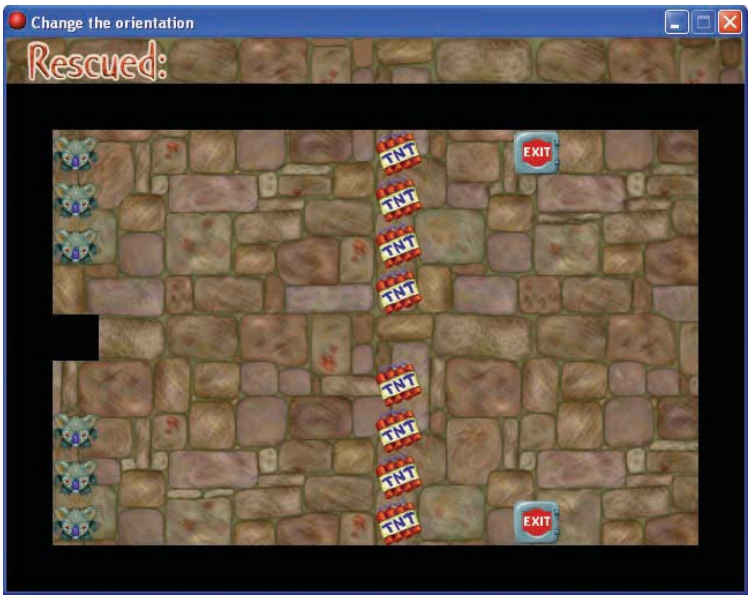

Figure 7-6. *This is a surprisingly difficult maze level.*

TNT is fun, but moving hazards should add an even greater challenge, so we're going to create two kinds of circular saws. One will move vertically and the other will move horizontally, but both will leave koalas wishing they had stayed in bed!

Creating the saw objects:

1. Create sprites called `spr_saw_horizontal` and `spr_saw_vertical` using `Saw_horizontal.gif` and `Saw_vertical.gif`. Enable the **Smooth edges** option for both.

2. Create an object called `obj_saw_horizontal` and give it the horizontal sprite. Add a **Create** event and include the **Move Fixed** action. Select the right arrow and set **Speed** to `4`. This is slightly slower than the speed of the koala to give the player a chance to escape.

3. Add a **Collision** event with the wall object and include the **Reverse Horizontal** action.

4. Add a **Collision** event with the koala object. Include the **Change Instance** action and select **Other** from **Applies to**, in order to change the koala. Set **Change into** to the dead koala object and set **Perform events** to **yes**.

5. Create an object called `obj_saw_vertical` and give it the vertical sprite. Add a **Create** event and include the **Move Fixed** action. Select the down arrow and set **Speed** to `4`.

6. Add a **Collision** event with the wall and include the **Reverse Vertical** action.

7. Finally add a **Collision** event with the koala object. Include the **Change Instance** action and select **Other** from **Applies to**. Set **Change into** to the dead koala object and set **Perform events** to **yes**.

At this point, you might be wondering why having the wall as a parent doesn't mess things up for the koala's collisions with saws. After all, if koalas are a "special kind of wall," then why don't the saws just turn themselves around when they collide with koalas? Fortunately Game Maker automatically chooses the most specific collision event and ignores the other (a koala is a koala first and only a wall second). However, if you remove the collision event between the saw and the koala, then saws will start treating koalas as if they were walls again!

Create some new levels using the moving saws. You might also want to add a cheat in the controller object, so that pressing the N key moves you to the next room and pressing the P key moves you to the previous room. You should know how to do this by now. You'll find the current version in `Games/Chapter07/koala4.gm6` on the CD.

Tiles

The walls of our maze need brightening up a bit, and we're going to do this by using *tiles*. Tiles work by creating a new background resource that consists of a number of small, identically sized images (40×40 pixels in our case). This is called a *tile set*, and we've created one that contains all the various combinations of wall connections that are needed to draw a maze (see Figure 7-7).

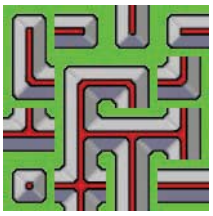

Figure 7-7. *This tile set contains 16 wall segments.*

Creating the tile set:

1. Create a new background called `back_tiles` using the file `Wall_tiles.bmp`. Enable the **Transparent** option so that the green areas appear transparent.
2. Enable the **Use as tile set** option. The properties form will then become larger to display all the properties of a tile set.
3. Set **Tile width** and **Tile height** to `40` and leave the other values as `0`. The image will now show an exploded view of the tiles, as shown in Figure 7-8.
4. Close the properties form.