SECOND LIFE®

scripting your world

the official guide to second life® scripting

By Dana Moore, Michael Thome, and Dr. Karen Zita Haigh

Foreword by Joe Miller, Linden Lab Vice President of Platform and Technology Development

SCRIPTING YOUR WORLD

THE OFFICIAL GUIDE TO SECOND LIFE® SCRIPTING

Dana Moore

Michael Thome

Dr. Karen Zita Haigh

Wiley Publishing, Inc.

Senior Acquisitions Editor: Willem Knibbe
Development Editor: Candace English
Technical Editor: Richard Platel
Production Editor: Patrick Cunningham
Copy Editor: Candace English
Production Manager: Tim Tate
Vice President and Executive Group Publisher: Richard Swadley
Vice President and Executive Publisher: Joseph B. Wikert
Vice President and Publisher: Neil Edde
Book Designer and Compositor: Patrick Cunningham
Proofreader and Indexer: Asha Johnson
Project Coordinator, Cover: Lynsey Stanford
Cover Designer: Ryan Sneed
Cover Image: Michael Thome

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-33983-1

Library of Congress Cataloging-in-Publication Data

```
Moore, Dana, 1947-
  Scripting your world: the official guide to second life scripting / Dana Moore, Michael Thome, Karen Haigh.
  -- 1st ed.
     p. cm.
  ISBN 978-0-470-33983-1 (paper/website)
1. Entertainment computing. 2. Scripting languages (Computer science) 3. Second Life (Game)
   I. Thome, Michael, 1965- II. Haigh, Karen, 1970- III. Title.
  QA76.9.E57M66 2008
  790.20285—dc22
                                                    2008027400
```


10 9 8 7 6 5 4 3 2 1

FOREWORD

In 2008 the virtual world of ***Second Life*** celebrated five years of existence. During that time, we have been participants in the creation of something truly extraordinary—a completely new realm in the Internet, a cyberspace unlike any other. Linden Lab and the residents of ***Second Life*** are the creative forces at work bringing this new metaverse, this three-dimensional world we can visit together via the Web, to life.

We think of ***Second Life*** as a place, although paradoxically, aside from a large set of servers and code, this new place has no physical existence outside the consensual agreement amongst a million human minds that here we dwell together in the three-dimensional Internet, the virtual world that is ***Second Life***. For the hundreds of thousands who visit here occasionally and for those who reside and do business here daily, this place of the mind exerts a pull as real as anywhere in the physical world. For the avant-garde thinkers, creators, and entrepreneurs who live at least a portion of their lives here, making tools, artifacts, environments, experiences, and even earning a living, this is our world.

Consisting of a series of sophisticated content-creation, land-management, transactional and scripting tools, the ***Second Life*** Grid is the technology platform used to power ***Second Life***. The Linden Scripting Language (LSL), embedded in all objects created in-world, enables object interactivity. LSL is a compact programming language made for virtual-world creation. Programmers and nonprogrammers alike are capable of creating in ***Second Life***; LSL is easy to learn and well-supported by in-world and real-world tools. With scripting, any element in ***Second Life*** can move, react, sense, change appearance and state. Without it, even the most detailed objects are akin to museum sculptures—inherently static.

In this Official Guide, thoroughly explained examples will prepare you to awaken objects' potential through scripting. Three seasoned software professionals coach the reader through their approach to understanding a new computer language for virtual creation in ***Second Life*** and on the Web.

As the new 3D world of ***Second Life*** begins to replace familiar software like the browser, the Java VM, or Windows, a new generation of software developers will inevitably emerge. This book is written with them in mind as well as the common ***Second Life*** resident, for the future extends beyond technical specifications and interface standards—it affects virtual objects, characters, and their interactions. We at Linden Lab believe this book serves as an educational travel companion for exploration of the innovative, virtual world.

Joe Miller,
Vice President of Technology and Platform Development

DEDICATION

*To all the amazingly creative people of **Second Life**, making and continuously remaking the world.*

ACKNOWLEDGMENTS

We could not have written this book without the assistance and support of many people, both in the real world and in ***Second Life***. First we need to thank our families for picking up the slack while we were immersed in writing: Kelly and Alicia Thome; Robert, Sonia, and Rachael Driskill; Jane and Caitlin Moore.

We would like to thank our reviewers, Aaron Helsinger, Robert Driskill, Trouble Streeter, and Roisin Hotaling for their time and patience. Also, many thanks to the scripting experts on the ***SL*** forums, scripting groups, mailing lists, and wikis—you might not ***know*** that you helped, but your conversations and documentation were invaluable in gaining the breadth of knowledge we needed to produce this book. Specific callouts to Strife Onizuka, Timeless Prototype, Delta Czukor, Adam Marker, Talin Sands, and Morrigan Mathilde.

Finally, a vigorous nod to our ***SL*** friends and cohorts, who have not only been interested and supportive during the writing process, but together have been a source of creativity, enthusiastic instigators for scripting projects, and tolerant models for screenshots. We would like to specifically thank Anu, Siv, Slade, Shel, Harper, Dillon, Amee, Jac, Midlou, Robert, and the Wellstone regulars. ElectricSheep would like to thank Morrigan for her friendship and conversations during the book-writing process.

Of course the book wouldn't be possible at all without the Sybex team. We would especially like to thank Willem Knibbe, our acquisitions editor, for investing the faith in this project, and of course our gratitude to the editorial and production staff for helping to produce a high quality work: Candace English, Patrick Cunningham, Pete Gaughan, and Richard Platel.

And no book on ***Second Life*** can fail to acknowledge the wizards behind Linden Lab, especially Philip Rosedale (Philip Linden) and Cory Ondrejka (Cory Linden).

ABOUT THE AUTHORS

Michael Thome (Vex Streeter) has been a computer scientist with BBN Technologies for over 20 years, and currently specializes in scalable computing technologies including parallel, distributed, and agent-based computing. Michael has degrees in cognitive science from the University of Rochester and Boston University, with specialties in computational models of human neurological systems. His interest in gaming started while he was in high school, building cellular automata systems to create terrains for the ***PSL Empire*** computer game, writing video games, and collaborating on one of the very first open source computer games, ***LSRHS Hack*** (the antecedent to the now-venerable ***NetHack***). His first ***Second Life*** avatar was rezzed in late 2006, but he's been a true active ***Second Life*** resident as Vex since only mid 2007. He currently lives in the Boston area with his real-life family, and also on a small desert island in Camembert.

Dr. Karen Zita Haigh's (Karzita Zabaleta) research is in machine learning for physical systems—she builds brains for robots. Karen has a Ph.D. in computer science from Carnegie Mellon University. She has worked in a variety of domains, including networking, cyber security, oil refineries, jet engines, and—far and away the most fun—the homes of elders. She was one of the analysts who looked at Space Shuttle ***Colombia***'s data after the explosion. Computer gaming in general never particularly interested her, but ***Second Life*** was different; it had just the right mix of elements—the ability to create content, stretch the limits of reality, and have fun without shooting everything she encountered. She was born in Kenya, lived in six countries, and holds citizenships in Canada, Great Britain, and the USA. Karen speaks fluent French and Mandarin Chinese. She lives in Minnesota with her husband and two daughters.

Dana Moore is a division scientist with BBN Technologies and is an acknowledged expert in the fields of peer-to-peer and collaborative computing, software agent frameworks, and sensor-imbued environments. Prior to joining BBN, Dana was Chief Scientist for Roku Technologies, and a Distinguished Member of Technical Staff at Bell Laboratories. Dana is a popular conference speaker and a university lecturer. He has written articles for numerous computing publications and published several books on topics ranging from peer-to-peer computing to rich Internet applications. Although not a gamer, Dana finds enormous value in the potential transformative value of ***Second Life***. Dana holds a Master of Science degree in engineering and a Bachelor of Science degree in industrial design from the University of Maryland. He lives in the Annapolis, Maryland, area with his family and also in a Tudor village in ***Second Life***.

INTRODUCTION

Learning the skills needed to activate and enliven ***Second Life*** has multiple benefits: You will be one of the rare folk able to lift the veil of mystery behind how things work in ***Second Life***. And you may be able to turn your talents and newly gained prowess into a revenue stream. Additionally, you will gain the joy of creating things that other ***Second Life*** residents will use and enjoy, and may well marvel at. But there's a far greater benefit to becoming a ***Second Life*** scripter ***par excellence***.

Years ago, most software-development challenges involved translating real-world actions, events, and objects into command-line arguments. Your creativity was severely tested due to the constraints and limits on both the computer and the user. Frankly, early interactive command-line applications were far from cool—in fact, they were exactingly tedious. This evolved into a long "Middle Ages" filled with desktop applications expressed in menus and windows—still a very frustrating experience from so many perspectives.

Skip ahead a few generations, and suddenly ***Second Life*** is on every pundit's radar. The overwhelming conclusion: ***Second Life*** begins to realize the promise of "cyberspace," a virtual place peopled with real humans interacting with virtual objects. Whether you are a hard-core software developer or simply an enthusiastic ***Second Life*** resident, this new 3D world gives you an opportunity to create things that operate like their real-world counterparts, or even more intriguingly, things that operate in ways that they never could in plain old reality. Scripting for ***Second Life*** affords you a whole new set of opportunities, bringing with it a whole new set of challenges and possibilities.

Second Life brings us new 3D development tools and, more importantly, a new 3D development perspective. The 3D development tools are both familiar (such as language constructs and events) and strange (such as coding the many possible interactions with an expressive, dynamic 3D object that responds to real-world inputs such as touch, gravity, proximity to other objects and avatars, and conversation). ***Second Life*** development perspectives are strangely familiar—all of us live our "first life" in a 3D world but until recently we painfully translated our surroundings to a flat, 2D computing world; think menus and windows. We humans are most comfortable living in 3D, and thanks in part the way that humans interact with computers and through computers to ***Second Life***, the future of computers and the Web will be an experience of sights, sounds, and sensation that are inherently 3D!

The emerging 3D Web will offer the benefits of useful, natural metaphors to interact, exchange information, run businesses, and the like. ***Second Life***—as the premiere 3D immersive web of people, places, and things—offers vast new opportunities. But it also creates development challenges that will demand a whole new set of skills and perspectives. This book gives you the insights, tools, and skills to take full advantage of this new 3D computing world by enabling you to build in ***Second Life***. We provide useful, working examples that you can implement and see in action in our 3D world.

Who Should Buy This Book

Most people, on becoming ***Second Life*** residents, spend their first few hours and days (once they've figured out how to maneuver and get to a sim of interest) socializing, dancing, buying sporty outfits, exploring alternate aspects of their personality, and doing sundry things analogous to their real-life activities. That's their primary filter for understanding what ***Second Life*** is and what it offers. Their first reaction is often, "Wow, look at all the neat things I can participate in and do. ***I wonder how many ways I can have fun?***"

As their comfort and experience level grows, many people begin to think, "Wow, look at all the nifty gadgets. ***I wonder how they work***?" Initially residents look at poofers and dance pads, jukeboxes and media players, windmills and waterfalls, admiring their functionality or their beauty; later these same people look

at the same artifacts and wonder how they were made and whether they could create a better version. In fact, in many cases they are ***convinced*** they could create a better version. If this description captures your experience even a little, congratulations! You've come to the right place. You are our audience.

As we wrote, we assumed the following about our readers:

- You are familiar with ***Second Life*** concepts. You don't need to be an expert, but you'll get lost pretty quickly if you haven't left Orientation Island.
- You understand basic ***SL*** building. Some of the scripting in this book requires some careful assembly of primitive objects, though more-complex examples are included in some of the other books in this series.
- You have played with scripts enough to know that you want to know more: you know how much scripting can add to the content you are creating. We don't assume you've attended a Scripting 101 class or read any scripting tutorials, but you'll have an easier time absorbing the book's ideas if you have.

You do not need to be a programmer, a mathematician, or a computer scientist—some parts of this book ***will*** be slow going if you haven't had any prior experience, but ***don't worry***! Nothing here is rocket science, and it is easy to skip over the parts that you aren't interested in or that seem a little too difficult. You can always come back to them later.

One of the great things about scripting in ***Second Life*** is that it is extremely easy to play with even advanced concepts and quickly grasp how things work: you won't hurt anything if you get it wrong the first or even the 47th time, but you'll learn a huge amount as you experiment.

Above all, enjoy the process!

What's Inside

Here is a glance at each chapter's offerings:

Chapter 1: Getting a Feel for the Linden Scripting Language begins the book by describing LSL and the basic concepts of ***Second Life*** scripting. It may be used both as an in-depth introduction for novices and as reference material for more-advanced scripters. We recommend that you at least skim Chapter 1 before diving into the chapters that interest you most.

Chapter 2: Making Your Avatar Stand Up and Stand Out includes some of the most basic and common scripted objects in ***Second Life***, with particular attention to scripting used to enhance avatars' appearance and behavior.

Chapter 3: Communications describes a variety of ways that scripts can communicate and interact with avatars and with other scripts.

Chapter 4: Making and Moving Objects covers various ways to create new objects and manipulate existing objects under script control.

Chapter 5: Sensing the World includes a variety of examples and projects that focus on using scripts to react automatically to their surroundings, including objects, avatars, and the environment.

Chapter 6: Land Design and Management illustrates how to manipulate the basic structure of the ***Second Life*** landscape, how to enable land security, and how to learn more about the land around you.

Chapter 7: Physics and Vehicles covers the ***Second Life*** simulation of physics, and the details of how to build basic vehicles. It includes a brief description of how to make flexiprims interact with physics.

Chapter 8: Inventory discusses how scripted objects can manipulate and manage their own inventory of objects, giving them to and accepting them from avatars and other objects.

Chapter 9: Special Effects uses the particle system, texture animation, and lighting to generate fireworks, fire, and lighting effects.

Chapter 10: Scripting the Environment describes scripting solutions to react to the *Second Life* environmental simulation, including wind, water, and time.

Chapter 11: Multimedia brings music, audio, video, and web content into the mix, describing how to present multimedia to *SL* users.

Chapter 12: Reaching Outside *Second Life* introduces communications between *SL* scripts and the outside world.

Chapter 13: Money Makes the World Go Round describes how to make your scripts deal with money, including building tip jars and vendor devices.

Chapter 14: Dealing with Problems discusses the sorts of scripting problems you are likely to see and their causes, and provides hints on how to fix them. It also discusses where and how to get and give help.

Chapter 15: New and Improved describes some newly implemented features and issues that scripters will be interested in, including the new Mono virtual machine.

Appendix A: Setting Primitive Parameters is a reference for the complexities of the `llPrimitiveParams()` family of functions. It describes the parameters themselves and what they do.

Appendix B: The Particle System details all of the options of the LSL function `llParticleSystem()`.

Appendix C: *SL* Community Standards lists the standards of behavior expected of *Second Life* residents.

Once you understand a given chapter's content, you should be well-equipped to attack any similar scripts. You can always go back to Chapter 1 to review, and the appendices are organized to be complete references.

We've written a whole lot more content than could be squeezed into the book. There are several bonus chapters, additional examples, and LSL resources that you can download from either the book's companion website (`http://syw.fabulo.us/`) or the publisher's website for the book at `http://www.sybex.com/WileyCDA/SybexTitle/productCd-0470339837.html`. The authors also maintain a headquarters inside *Second Life* at Hennepin <38,136,108>; you can also find it by searching for SYWHQ, where you can see and get demonstrations of just about everything in this book and the components on the website.

BUILD NOTE

Build Notes (which look like this!) contain special instructions for how to build the objects that go with the nearby scripts whenever special tricks or building pitfalls are present.

How to Contact the Authors

We welcome feedback from you about this book or about books you'd like to see from us in the future. You can reach us by writing to `authors@syw.fabulo.us`.

Sybex strives to keep you supplied with the latest tools and information you need for your work. Please check their website at `www.sybex.com`, where we'll post additional content and updates that supplement this book should the need arise. Enter ***`Scripting Your World`*** in the Search box (or type the book's ISBN—**9780470339831**), and click Go to get to the book's update page.

CONTENTS

CHAPTER 1

GETTING A FEEL FOR THE LINDEN SCRIPTING LANGUAGE

What is so compelling about ***Second Life***? As Linden Lab Founder Philip Rosedale explained in an October 19, 2006 interview in ***The New York Times***, in ***Second Life*** avatars can move around and do everything they do in the real world, but without constraints such as the laws of physics: "When you are at Amazon.com [using current web technology] you are actually there with 10,000 concurrent other people, but you cannot see them or talk to them," Rosedale said. "At ***Second Life***, everything you experience is inherently experienced with others."

Much of the reason Rosedale can talk convincingly about shared experience is because of scripting in ***Second Life***. To be sure, ***Second Life*** is a place of great physical beauty: in a well-crafted ***SL*** environment, a feeling of mood and cohesive appearance lend a level of credibility to the experience of existing and interacting in a specific and sometimes unique context. But consider how sterile ***Second Life*** would be without scripting. Builders and artists create beautiful vistas, but without interaction the world is ***static***; little more than a fancy backdrop. Scripts give the world ***life***, they allow avatars to be more realistic, and they enhance the residents' ability to react to and interact with each other and the environment, whether making love or making war, snorkeling, or just offering a cup of java to a new friend.

This chapter covers essential elements of scripting and script structure. It is intended to be a guide, and may be a review for you; if that's the case then skim it for nuggets that enhance your understanding. If you are new to ***Second Life*** scripting or even programming in general, consider this chapter an introduction to the weird, wonderful world of ***SL*** scripting and the Linden Scripting Language, LSL. If you don't understand something, ***don't worry***! You might find it easier to skip ahead and return here to get the details later.

NOTE

Throughout the book, you'll see references to the *LSL wiki*. There are actually several such wikis, of which `http://wiki.secondlife.com/wiki/LSL_Portal` is the "official" one, and `http://lslwiki.net` is one of many unofficial ones. Typing out the full URL is cumbersome and hard to read, so if you see a reference to the wiki, you'll see only the keyword. For example, if you see, "you'll find more about the particle system on the LSL wiki at llParticleSystem," it means `http://wiki.secondlife.com/wiki/LlParticleSystem`, `http://www.lslwiki.net/lslwiki/wakka.php?wakka=llParticleSystem` or `http://rpgstats.com/wiki/index.php?title=LlParticleSystem`. All of these wikis have search functions and convenient indexes of topics.

In general, all examples in this book are available at the *Scripting Your World* Headquarters (SYW HQ) in *Second Life* at Hennepin <38, 136, 108>* and on the Internet at `http://syw.fabulo.us`. There are also several extras that didn't get included in the book due to space limitations. Enjoy browsing!

---

* Visit `http://slurl.com/secondlife/Hennepin/38/138/108/` or simply search in-world for "SYWHQ."

SCRIPTING STRUCTURE 101

A script is a ***Second Life*** asset, much like a notecard or any other Inventory item. When it is placed in a ***prim***, one of the building blocks of all simulated physical objects, it can control that prim's behavior, appearance, and relationship with other prims it is linked with, allowing it to move; change shape, color, or texture; or interact with the world.

NOTE

A *prim* is the basic primitive building block of *SL*: things like cubes, spheres, and cylinders. An *object* is a set of one or more linked prims. When you link the prims, the *root* prim is the one that was selected last; the remaining prims are called *children*. The root prim acts as the main reference point for every other prim in the object, such as the name of the object and where it attaches.

Whether or not you've already begun exploring how to script, you've probably created a new object and then clicked the New Script button. The result is that a script with no real functionality is added to the prim's Content folder. Left-clicking on the script opens it in the in-world editor and you see the script shown in Listing 1.1. It prints `"Hello, Avatar!"` to your chat window and then prints `"Touched."` each time you click the object that's holding it.

Listing 1.1: Default New Script

```
default
{
    state_entry()
    {
        llSay(0, "Hello, Avatar!");
    }

    touch_start(integer total_number)
    {
        llSay(0, "Touched.");
    }
}
```

This simple script points out some elements of script structure. First of all, scripting in ***SL*** is done in the ***Linden Scripting Language***, usually referred to as ***LSL***. It has a syntax similar to the common C or Java programming languages, and is event-driven, meaning that the flow of the program is determined by events such as receiving messages, collisions with other objects, or user actions. LSL has an explicit ***state model*** and it models scripts as finite state machines, meaning that different classes of behaviors can be captured in separate states, and there are explicit transitions between the states. The state model is described in more detail in the section "States" later in this chapter. LSL has some unusual built-in data types, such as vectors and quaternions, as well as a wide variety of functions for manipulating the simulation of the physical world, for interacting with player avatars, and for communicating with the real world beyond ***SL***.

The following list describes a few of the key characteristics of an LSL script. If you're new to programming, don't worry if it doesn't make much sense just yet; the rest of this chapter explains it all in more detail. The section "An LSL Style Guide" ties things together again.

- All statements must be terminated by a semicolon (`;`).
- LSL is block-oriented, where blocks of associated functionality are delimited by opening and closing curly braces ({ ***block*** }).
- Variables are typed and declared explicitly: you must ***always*** specify exactly which type a variable is going to be, such as `string` or `integer`.
- At a bare minimum, a script must contain the `default` state, which must define at least one ***event handler,*** a subroutine that handles inputs received in a program, such as messages from other objects or avatars, or sensor signals.
- Scripts may contain user-defined functions and global variables.

Listing 1.2 shows a rather more complete script, annotated to point out other structural features. This script controls the flashing neon sign on the front of the ***Scripting Your World*** visitor center. **Do not be discouraged if you don't understand what is going on here!** Although this script is relatively complex, it is here to illustrate that you don't ***need*** to understand the details to see how a script is put together.

This first discussion won't focus on the function of the neon sign, but rather on the structure commonly seen in LSL scripts. A script contains four parts, generally in the following order:

- ***Constants*** (colored orange in Listing 1.2)
- ***Variables*** (green)
- ***Functions*** (purple)
- ***States***, starting with `default` (light blue, with the event handlers that make a state in dark blue)

While common convention uses this order for constants, variables, and user-defined functions, they are permitted to occur in any order. They ***must*** all be defined before the `default` state, however. Additionally, you are required to have the `default` state before any user-defined states.

NOTE

Constants are values that are never expected to change during the script. Some constants are true for all scripts, and part of the LSL standard, including `PI` (3.141592653), `TRUE` (1), and `STATUS_PHYSICS` (which indicates whether the object is subject to the *Second Life* laws of physics). You can create named constants for your script; examples might include `TIMER_INTERVAL` (a rate at which a timer should fire), `COMMS_CHANNEL` (a numbered communications channel), or `ACCESS_LIST` (a list of avatars with permission to use the object).

Variables, meanwhile, provide temporary storage for working values. Examples might include the name of the avatar who touched an object, counts of items seen, or the current position of an object. The section "Variables" later in this chapter describes variables in more detail.

Functions are a mechanism for programmers to break their code up into smaller, more manageable chunks that do specific subtasks. They increase readability of the code and allow the programmer to reuse the same capability in multiple places. The section "User-Defined Functions" describes functions in more detail.

Listing 1.2: Flipping Textures by Chat and by Timer

COMMENTS | CONSTANTS | VARIABLES | FUNCTIONS | STATES | EVENT HANDLERS

```
// Texture Flipper for a neon sign

// Constants
integer TIMER_INTERVAL   = 2; // timer interval
string  NEON_OFF_TEXTURE = "bcf8cd82-f8eb-00c6-9d61-e610566f81c5";
string  NEON_ON_TEXTURE  = "6ee46522-5c60-c107-200b-ecb6e037293e";

// global variables
integer gOn    = TRUE;          // If the neon is burning
integer gSide = 0;              // which side to flip
integer gListenChannel = 989; // control channel

// functions
fliptexture(string texture) {
   llSetTexture(texture, gSide);
}

usage(){
   llOwnerSay("Turn on by saying: /"+(string)gListenChannel+" sign-on");
   llOwnerSay("Turn off by saying: /"+(string)gListenChannel+" sign-off");
}

// states
default
{
    state_entry() {
        llSetTimerEvent(TIMER_INTERVAL);
        llListen( gListenChannel, "", llGetOwner(), "" );
    }

    listen(integer channel, string name, key id, string msg) {
        if (msg == "sign-on") {
            fliptexture(NEON_ON_TEXTURE);
            gOn = TRUE;
            llSetTimerEvent(TIMER_INTERVAL); // start the timer
        } else if (msg == "sign-off") {
            fliptexture(NEON_OFF_TEXTURE);   // start the timer
            gOn = FALSE;
            llSetTimerEvent(0.0);
        } else {
            usage();
        }
    }

    timer() {
        if (gOn){
            fliptexture(NEON_OFF_TEXTURE);
            gOn = FALSE;
        } else {
            fliptexture(NEON_ON_TEXTURE);
            gOn = TRUE;
        }
    }
}
```

Two forward slashes (`//`) indicate a ***comment***. The slashes and the entire rest of the line are completely ignored by ***SL***. They remain part of the script but have no effect, so you can use them to add a copyright notice or a description of what's going on in the script, or even to disable lines when you are trying to debug a problem. Likewise, empty lines and extra spaces play no part in the execution of a script: indentation helps readability but ***SL*** ignores it.

Declarations of global constants and variables have script-wide scope; that is, the entire rest of the script can use them. Most programmers are taught that global variables are evil, but in LSL there is no other way to communicate information between states. Since most LSL scripts are fairly short, it's relatively easy to keep track of these beasties, eliminating one of the major reasons that global variables are discouraged in other languages. Although technically the LSL compiler does not distinguish between user-defined constants and variables, the examples in this book name constants with all capital letters, and global variables using mixed case beginning with the lowercase letter ***g***.

Next you will notice a couple of code segments that seem to be major structural elements; these are called `fliptexture()` and `usage()`, respectively. These are user-defined functions. Functions are global in scope and available to all states, event handlers, and other user-defined functions in the same script. Functions can return values with a `return` command. The "Functions" section in this chapter provides considerably more detail. Linden Library functions are readily identifiable, as they (without exception) begin with the letters `ll`, as in `llSetTimerEvent()`.

The last elements of a script are the ***states***. A state is a functional description of how the script should react to the world. For example, you could think of a car being in one of two states: when it is on the engine is running, it is making noises, it can move, it can be driven. When it is off it is little more than a hunk of metal; it is quiet, immobile, and cannot be driven. An LSL script represents an object's state of being by describing how it should react to events in each situation. Every script must have at least the one state, `default`, describing how it behaves, but you can define more states if it makes sense. An ***event*** is a signal from the ***Second Life*** simulation that something has happened to the object, for example that it has moved, been given money, or been touched by an avatar. When an event happens to a ***Second Life*** object, each script in the object is told to run the matching ***event handler***: As an example, when an avatar touches an object, ***SL*** will run the `touch_start()`, `touch()`, and `touch_end()` event handlers in the active state of each script in the object, if the active state has those handlers. LSL has defined a set number of event handlers. (The SYW website has a complete list of event handlers and how they are used.) The three event handlers in Listing 1.2, `state_entry()`, `listen()`, and `timer()`, execute in a finite state machine managed by the simulator in which the object exists. More details on the state model are presented in the section "States," as it is one of the more interesting aspects of LSL.

You may well ask, "So what does this script do?" It's really pretty simple. Whenever a couple of seconds tick off the clock (the time interval defined by the constant `TIMER_INTERVAL`), the timer event fires, and the texture on the front face of the object is replaced either with the "on" texture referenced by the key in the string `NEON_ON_TEXTURE` or with the "off" texture, `NEON_OFF_TEXTURE`. The script also listens for input by anyone who knows the secret channel to talk to the object (989, declared as the variable ***`gListenChannel`***). If the object hears anyone chat ***`sign-off`*** or ***`sign-on`*** on the secret channel*, it will activate or deactivate the sign. Come by SYW HQ and tell our sign to turn off (or on, as the case may be). Figure 1.1 shows the script in action. Chapter 3, "Communications," talks more about channels and how to communicate with objects.

* When typing in the chat window, the channel number is preceded by a slash, as in ***`/989 sign-off.`***

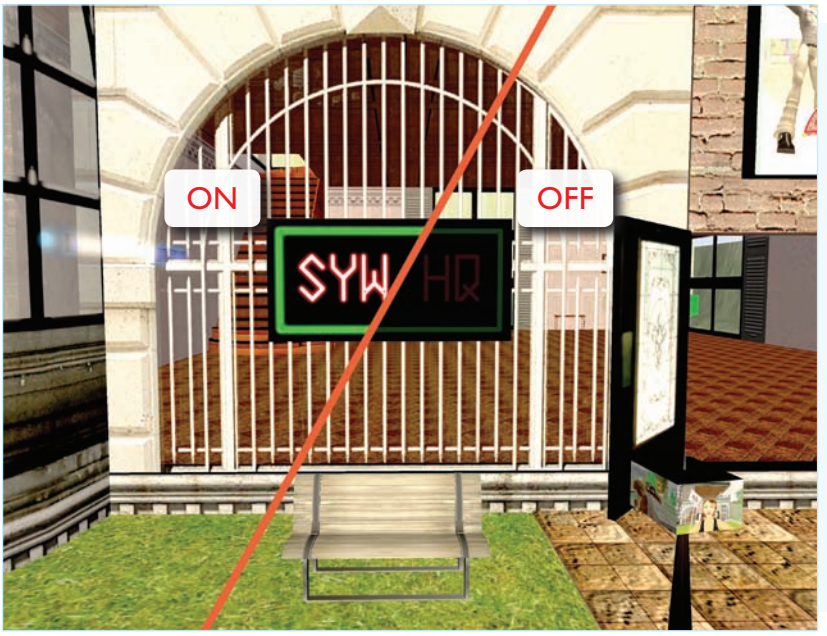

Figure 1.1: Texture-flipping in action

A ***type*** is a label on a piece of data that tells the computer (and the programmer) something about what kind of data is being represented. Common data types include integers, floating-point numbers, and alphanumeric strings. If you are familiar with the C family of languages (C, C++, C#, Java, and JavaScript) you'll notice similarities between at least a few of them and LSL. Table 1.1 summarizes the valid LSL variable types. Different data types have different constraints about what operations may be performed on them. Operators in general are covered in the "Operators" section of this chapter, and some of the sections that cover specific types also mention valid operations.

Because all variables in LSL are typed, type coercion is awkward. Most coercion must be done manually with ***explicit casting***, as in

```
integer i=5;
llOwnerSay((string)i);
```

In many cases, LSL does the "right thing" when coercing (***implicit casting***) types. Almost everything can be successfully cast into a string, integers and floats are usually interchangeable, and other conversions usually result in the null or zero-equivalent. The discussion later, in Table 1.7, of `llList2<type>()` functions gives a good overview of what happens. Look on the Types page of the LSL wiki for an expanded example of coercion.

TABLE 1.1: LSL VARIABLE TYPES

Data Type	Usage
`integer`	Whole number in the range –2,147,483,648 to 2,147,483,647.
`float`	Decimal number in the range 1.17549435E-38 to 3.402823466E+38.
`vector`	A three-dimensional structure in the form `<x, y, z>`, where each component is a float. Used to describe values such as a position, a color, or a direction.
`rotation quaternion`	A four-dimensional structure consisting of four floats, in the form `<x, y, z, s>`, that is the natural way to represent rotations. Also known as a ***quaternion***, the two type names are interchangeable.
`key`	A UUID (specialized string) used to identify something in ***SL***, notably an agent, object, sound, texture, other inventory item, or data-server request.
`string`	A sequence of characters, limited only by the amount of free memory available to the script (although many functions have limits on the size they will accept or return).
`list`	A heterogeneous collection of values of any of the other data types, for instance `[1, "Hello", 4.5]`.

NOTE

The value of a variable will never change unless your code reassigns it, either explicitly with = or implicitly with an operator such as ++, which includes a reassignment:

```
a = "xyzzy";
b = a;          // b is also "xyzzy"
a = "plugh";    // a is now "plugh" but b is still "xyzzy"!
```

This holds true for all types, including lists! In keeping with this immutability, all function parameters are pass-by-value (meaning only the value is sent) in LSL.

INTEGER

Integers are signed (positive or negative) 32-bit whole numbers. LSL does not provide any of the common variations on integer types offered in most other languages. Integers are also used to represent a few specific things in LSL:

- ***Channels*** are integer values used to communicate in "chat" between both objects and avatars. See the section "Talking to an Object (and Having It Listen)" in Chapter 3 for a deeper discussion of channels and their use.
- ***Booleans*** are implemented as integer types with either of the constant values: `TRUE` (1) or `FALSE` (0).
- ***Event counters*** are integer arguments to event handlers that indicate how many events are pending. Inside such an event handler, the `llDetected*()` family of library functions can be used to determine which avatars touched an object, which other objects collided with yours, or which objects are nearby.
- ***Listen handles*** are returned by `llListen()` and enable code to have explicit control over the listen stream. (Other things you might think would be handles are actually returned as keys.) See Chapter 2, "Making Your Avatar Stand Up and Stand Out," for examples of `llListen()`.

- ***Bit patterns*** (or ***bit fields***) are single integers that represent a whole set of Boolean values at once. Different bits can be combined to let you specify more than one option or fact at once. For instance, in the `llParticleSystem()` library function, you can indicate that particles should bounce and drift with the wind by combining the constant values `PSYS_PART_BOUNCE_MASK` and `PSYS_PART_WIND_MASK` by saying

```
PSYS_PART_BOUNCE_MASK | PSYS_PART_WIND_MASK
```

FLOAT

A `float` in LSL is a 32-bit floating-point value ranging from ±1.401298464E–45 to ±3.402823466E+38. Floats can be written as numbers, such as 1.0 or 9.999, and they can be written in scientific notation, as in 1.234E–2 or 3.4E+38, meaning 1.234×10^{-2} and 3.4×10^{38}.

A `float` has a 24-bit signed ***mantissa*** (the number), and an 8-bit signed ***exponent***. Thus for a float 1.2345E+23, the number 1.2345 is the mantissa, and 23 is the exponent.

Because one bit represents the sign of the number (positive or negative), a 23-bit mantissa gives a precision equivalent to approximately 7 decimal digits—more precisely $\log_{10}(2^{23})$. This means values are rarely stored exactly. For example, if you do something like

```
float foo = 101.101101;
```

and print the result, it will report 101.101105, so you should expect some rounding inaccuracy. Even 10E6 × 10E6 isn't 10E12, instead printing 100000000376832.000000. Often more disturbingly, addition or subtraction of two numbers of vastly different magnitudes might yield unexpected results, as the mantissa can't hold all the significant digits.

When an operation yields a number that is too big to fit into a float, or when it yields something that is not a number (such as `1.0 / 0.0`), your script will generate a run-time Math Error.

VECTOR

Vectors are ***the*** currency of three-dimensional environments, and so are found throughout LSL code. In addition, anything that can be expressed as a triplet of `float` values is expressed in a `vector` type. If you were guessing about the kinds of concepts readily expressed by a set of three values, you'd probably come up with positioning and color, but there are also others, shown in Table 1.2.

TABLE 1.2: COMMON USES FOR VECTORS AND WHAT THEY REPRESENT

Vector Concept	What Vector Represents
Position	Meters. Always relative to some base positioning (the sim, the avatar, or the root prim).
Size	Meters, sometimes also called scale.
Color	Red, green, blue. Each component is interpreted in a range from 0.0 to 1.0; thus yellow is `vector yellow = <1.0, 1.0, 0.0>;`
Direction	Unitless. It is usually a good idea to normalize directions (see `llVecNorm()`); since directions are often multiplied with other values, non-unit direction vectors can have an unexpected proportional effect on the results of such operations.
Velocity	An offset from a position in meters traveled per second. You can also think of velocity as a combination of direction and speed in meters per second.
Acceleration	Meters per second squared.
Impulse	Force (mass × velocity).
Rotation	Radians of yaw, pitch, and roll. Also known formally as the ***Euler*** form of a rotation.

The x, y, and z components of a vector are floats, and therefore it is slightly more efficient to write a vector with float components—for instance, <0.0, -1.0, 123.0>—than with integer components. Here are some examples of ways to access vectors, including the built-in constant ZERO_VECTOR for <0.0,0.0,0.0>:

```
vector aVector = <1.0, 2.0, 3.0>;
float xPart = 1.0;
vector myVec = <xPart, 2.0, 3.0>;
float yPart = myVec.y;
float zPart = myVec.z;
myVec.y = 0.0;
vector zeroVec = ZERO_VECTOR;
llOwnerSay("The empty vector is "+(string)ZERO_VECTOR);
```

Vectors may be operated on by scalar floats (regular numbers); for instance, you could convert the yellow color vector in Table 1.2 to use the Internet-standard component ranges of 0 to 255 with the expression <1.0, 1.0, 0.0>*255.0. Vector pairs may be transformed through addition, subtraction, vector dot product, and vector cross product. Table 1.3 shows the results of various operations on two vectors:

```
vector a = <1.0, 2.0, 3.0>;
vector b = <-1.0, 10.0, 100.0>;
```

TABLE 1.3: MATHEMATICAL OPERATIONS ON VECTORS

Operation	Meaning	Vector
+	Add	a+b = < 0.0, 12.0, 103.0 >
-	Subtract	a-b = < 2.0, -8.0, -97.0>
*	Vector dot product	a*b = 319.0 (1 * -1) + (2 * 10) + (3 * 100)
%	Vector cross product	a%b = < 170.0, -103.0, 12.0 > <(2 * 100) - (3 * 10), (3 * -1) - (1 * 100), (1 * 10) - (2 * -1) >

Coordinates in *SL* can be confusing. There are three coordinate systems in common use, and no particular annotation about which is being used at any given time.

- **Global coordinates**. A location anywhere on the *Second Life* grid with a unique vector. While not often used, every place on the grid has a single unique vector value when represented in global coordinates. Useful functions that return global coordinates include llGetRegionCorner() and llRequestSimulatorData().
- **Region coordinates**. A location that is relative to the southwest corner of the enclosing sim (eastward is increasing x, northward is increasing y, up is increasing z), so the southwest corner of a sim at altitude 0 is <0.0, 0.0, 0.0>. The position or orientation of objects, when not attached to other prims or the avatar, is usually expressed in terms of regional coordinates.

  A region coordinate can be converted to a global coordinate by adding to it the region corner of the simulator the coordinate is relative to:

```
vector currentGlobalPos = llGetRegionCorner() + llGetPos();
```

- **Local coordinates**. A location relative to whatever the object is attached to. For an object in a linkset, that means relative to the root prim. For an object attached to the avatar, that means relative to the avatar. For the root prim of the linkset, that value is relative to the sim (and therefore the same as the region coordinates). If the attachment point moves (e.g., the avatar moves or the root prim rotates), the object will move relative to the attachment, even though local coordinates do not change. For example, if an avatar moves her arm, her bracelet will stay attached to her wrist; the bracelet is still the same distance from the wrist, but not in the same place in the region.

Useful functions on vectors include `llVecMag(vector v)`, `llVecNorm(vector v)`, and `llVecDist(vector v1, vector v2)`. `llVecMag()` calculates the magnitude, or length, of a vector—it's Pythagoras in three dimensions. These functions are really useful when measuring the distance between two objects, figuring out the strength of the wind or calculating the speed of an object. `llVecNorm()` normalizes a vector, turning it into a vector that points in the same direction but with a length of 1.0. The result can be multiplied by the magnitude to get the original vector back. `llVecNorm()` is useful for calculating direction, since the result is the simplest form of the vector. `llVecDist(v1,v2)` returns the distance between two vectors ***v1*** and ***v2***, and is equivalent to `llVecMag(v1-v2)`.

ROTATION

There are two ways to represent rotations in LSL. The native rotation type is a ***quaternion***, a four-dimensional vector of which the first three dimensions are the axes of rotation and the fourth represents the angle of rotation. `quaternion` and `rotation` can be used interchangeably in LSL, though `rotation` is much more common.

Also used are ***Euler*** rotations, which capture yaw (`x`), pitch (`y`), and roll (`z`) as `vector` types rather than as `rotation` types. The LSL Object Editor shows rotations in Euler notation. Euler notation in the Object Editor uses degrees, while quaternions are represented in radians; a circle has 360 degrees or `TWO_PI` (6.283) radians.

Euler vectors are often more convenient for human use, but quaternions are more straightforward to combine and manipulate and do not exhibit the odd discontinuities that arise when using Euler representation. For instance, in the ***SL*** build tools, small changes in object rotation can make sudden radical changes in the values indicated. Two functions convert Euler representations into quaternions (and vice versa): `llEuler2Rot(vector eulerVec)` and `llRot2Euler(rotation quatRot)`. Many of your scripts can probably get away with never explicitly thinking about the guts of quaternions:

```
// convert the degrees to radians, then convert that
// vector into a quaternion
rotation myQuatRot = llEuler2Rot(<45.0, 0.0, 0.0> * DEG_TO_RAD);
// convert the rotation back to a vector
// (the values will be in radians)
vector myEulerVec = llRot2Euler(myQuatRot);
```

The above code snippet converts the degrees to radians by multiplying the vector by `DEG_TO_RAD`. Two other constants—`ZERO_ROTATION` and `RAD_TO_DEG`—are useful for rotations. These constants are defined in Table 1.4.

TABLE 1.4: CONSTANTS USEFUL FOR MANIPULATING ROTATIONS

Constant	Value	Description
`ZERO_ROTATION`	<0.0, 0.0, 0.0, 1.0>	A `rotation` constant representing a Euler angle of <0.0, 0.0, 0.0>.
`DEG_TO_RAD`	0.01745329238	A `float` constant that, when multiplied by an angle in degrees, gives the angle in radians.
`RAD_TO_DEG`	57.29578	A `float` constant that, when multiplied by an angle in radians, gives the angle in degrees.

You will find much more in-depth discussion and some examples for using rotations in Chapter 4, "Making and Moving Objects."

KEY

A ***key*** is a distinctly typed string holding the UUID for any of a variety of relatively long-lived ***SL*** entities. A UUID, or Universal Unique Identifier, is a 128-bit number assigned to any asset in ***Second Life***, including avatars, objects, and notecards. It is represented as a string of hex numbers in the format `"00000000-0000-0000-0000-000000000000"`, as in `"32ae0409-83d6-97f5-80ff-6bee5f322f14"`. `NULL_KEY` is the all-zero key. A `key` uniquely identifies each and every long-lived item in ***Second Life***.

In addition to the unsurprising use of keys to reference assets, keys are also used any time your script needs to request information from a computer other than the one it is actually running on, for instance to web servers or to the ***SL*** dataserver, to retrieve detailed information about ***SL*** assets. In these situations, the script issues a request and receives an event when the response is waiting. This model is used to ask not just about avatars using the `llRequest*Data()` functions, but also to do things like read the contents of notecards with `llGetNotecardLine()`. Asynchronous interactions with the outside world might include HTTP requests, `llHTTPRequest()`, and are identified with keys so that the responses can be matched with the queries.

Numerous LSL functions involve the manipulation of keys. Some of the main ones are shown in Table 1.5.

TABLE 1.5: SAMPLE FUNCTIONS THAT USE KEYS

Function Name	Purpose
`key llGetKey()`	Returns the key of the prim.
`key llGetCreator()`	Returns the key of the creator of the prim.
`key llGetOwner()`	Returns the key of the script owner.
`key llDetectedKey()`	Returns the key of the sensed object.
`string llKey2Name(key id)`	Returns the name of the object whose key is ***id***.
`key llGetOwnerKey(key id)`	Returns a key that is the owner of the object ***id***.

Note that there is no built-in function to look up the key for a named avatar who is not online. However, there are a number of ways to get this information through services, such as `llRequestAgentData()`.