
COPYRIGHTED MATERIAL

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.`***

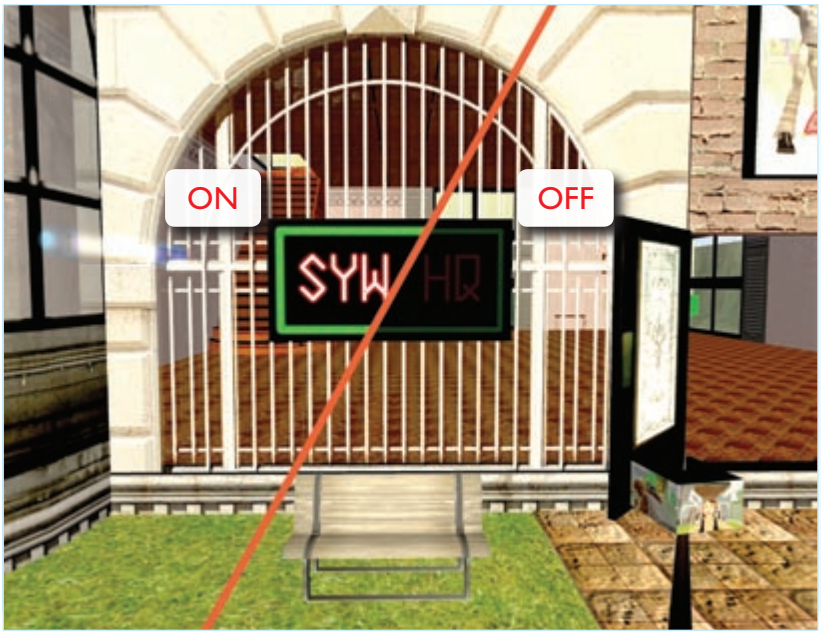

Figure 1.1: Texture-flipping in action

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 1.175494351E-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()`.

STRING

Strings are sequences of letters, numerals, and other characters, collected as a single value. You can specify a constant string in LSL by surrounding the sequence of characters you want with double quotes ("). You can include a double-quote symbol in a string by prefixing it with a backslash (\). Similarly, you can include a backslash with \\, a sequence of four spaces with \t, and a newline with \n. Table 1.6 shows the set of string-manipulation functions provided by LSL.

String indexes are zero-based, ranges are always inclusive of the start and end index, and negative indexes indicate counting positions backwards from the end of the string (-1 is the last character of the string). Consider this example:

```
string test = "Hello world! ";
llGetSubString(test,0,0);    // "H"
llGetSubString(test,0,-1);   // "Hello world! "
llGetSubString(test, -6,-2); // "world"
```

String values, like all other LSL values, are ***immutable***; that is, no function will modify the original string. Rather, functions return brand-new strings that contain transformed versions. For example, the variable ***test*** in the following snippet does not change when a word is inserted into the string:

```
string test = "Hello world! ";
string x = llInsertString(test, 6, "cruel ");
// x = "Hello cruel world! ", test is unchanged
```

TABLE 1.6: FUNCTIONS THAT MANIPULATE STRINGS

Function Name	Purpose
`integer llStringLength(string s)`	Gets the length of a string.
`string llToLower(string s)`	Converts a string to lowercase.
`string llToUpper(string s)`	Converts a string to uppercase.
`integer llSubStringIndex(string s, string pattern)`	Finds the integer position of a string in another string.
`string llGetSubString(string s, integer start, integer end)`	Extracts a part of a string; returns the part.
`string llDeleteSubString(string s, integer start, integer end)`	Returns a copy of the original minus the specified part.
`string llInsertString(string s, integer pos, string snippet)`	Inserts a string snippet into a string starting at the specified position.
`string llStringTrim(string s, integer trimType)`	Trims leading and/or trailing whitespace from a string. Trim types are `STRING_TRIM_HEAD` for the leading spaces, `STRING_TRIM_TAIL` for the trailing spaces, and `STRING_TRIM` for both leading and trailing spaces.
`string llEscapeURL(string url)`	Returns the string that is the URL-escaped version of ***url*** (replacing spaces with `%20`, etc).
`string llUnescapeURL(string url)`	Returns the string that is the URL unescaped version of ***url*** (replacing `%20` with spaces, etc).

LIST

Lists in LSL are collections of values of any other types. Lists are ***heterogeneous***: they may contain any mixture of values of any type except other lists. This makes it important to keep track of what is stored in your lists to avoid getting into trouble, but you can type-check elements at runtime if you need to. For instance, the following code extracts elements from a list:

```
list mylist = [1, 2.3, "w00t", <1.0, 0.0, 0.0>];
integer count = llList2Integer(mylist, 0);
float value = llList2Float(mylist, 1);
string exclamation = llList2String(mylist, 2);
vector color = llList2Vector(mylist, 3);
```

To access the individual elements, use the `llList2<type>()` functions, shown in Table 1.7.

You can use `llGetListEntryType()` to find out the type of the element. For example, `llList2Float(list src, integer index);` returns the float value at the specified index in the list. Thus Listing 1.3 prints 4.000000 when the object is touched. Note that it is implicitly casting the original `integer` value into a `float`.

TABLE 1.7: FUNCTIONS THAT EXTRACT INDIVIDUAL LIST ELEMENTS

Function Name	Purpose
`string llList2String(list l, integer index)`	Returns a string element. All other types are appropriately coerced into a string.
`integer llList2Integer(list l, integer index)`	Returns an integer element. If the element is not coercible to an integer, it will return 0.
`float llList2Float(list l, integer index)`	Returns a float element. If the element is not coercible to a float, it will return 0.0.
`key llList2Key(list l, integer index)`	Returns a key element. If the element is a regular string, it is returned unchanged. Other non-key elements return `NULL_KEY`.
`vector llList2Vector(list l, integer index)`	Returns a vector element. If the element is not coercible to a vector, it will return `ZERO_VECTOR`.
`rotation llList2Rot(list l, integer index)`	Returns a rotation element. If the element is not coercible to a rotation, it will return `ZERO_ROTATION`.
`integer llGetListEntryType(list l, integer index)`	Gets the type of entry of an element in the list. Returns an integer constant `TYPE_INTEGER`, `TYPE_FLOAT`, `TYPE_STRING`, `TYPE_KEY`, `TYPE_VECTOR`, `TYPE_ROTATION`, or `TYPE_INVALID`. Invalid occurs when the index was out of range.

Listing 1.3: Coercing an Element of a List into a Float

```
list gMyNumList=[1,2,3,4,5];
default
{
    touch_start(integer total_number) {
        float f = llList2Float(gMyNumList,3);
        llOwnerSay((string)f);
    }
}
```

The upside of heterogeneous lists in the LSL context is that you can imitate ***associative arrays*** (for instance, dictionaries or reference tables) with careful searches. If you keep your lists well-structured, you're very likely to know your list contents and their types very well, even when the contents are a mixed bag. The first element in a list is at index 0, and negative indexes reference elements counting backwards from the end of the list, with `-1` indicating the last one. Table 1.8 shows some of the basic list-manipulation functions.

TABLE 1.8: SOME CORE FUNCTIONS TO MANIPULATE LISTS

Function Name	Purpose
`integer llGetListLength(list l)`	Gets the number of elements in a list.
`integer llListFindList(list l, list test)`	Returns the index of the first instance of ***test*** in ***l***, or `-1` if it isn't found.
`list llList2List(list l, integer start, integer end)`	Returns the portion of a list, with the specified elements (similar to substring).
`list llDeleteSubList(list l, integer start, integer end)`	Removes a portion of a list, returns the list minus the specified elements.
`list llListInsertList(list dest, list snippet, integer pos)`	Inserts a list into a list. If ***pos*** is longer than the length of ***dest***, it appends the snippet to the ***dest***.
`list llListReplaceList(list dest, list snippet, integer start, integer end)`	Replaces a part of a list with another list. If the snippet is longer than the ***end-start***, then it serves to insert elements too.
`float llListStatistics(integer operation, list input)`	Performs statistical operations, such as min, max, and standard deviation, on a list composed of integers and floats. The ***operation*** takes one of ten `LIST_STAT_*` constants, which are described fully on the SYW website.

You can use + and += as concatenation operators:

```
list a = [1,2];
a = a + [3];    // a now contains [1, 2, 3]
a += [4];       // a now contains [1, 2, 3, 4]
```

While you can't make nested lists, you can insert a list into another:

```
list myL = ["a", "b"];
list myLL = llListInsertList(myL,[1,2,3,4],1);
// myLL contains [a, 1, 2, 3, 4, b]
```

Perhaps the most interesting set of string functions deals with converting back and forth between lists and strings, listed in Table 1.9. For example, the function `llDumpList2String()` concatenates the items in a list into a string with a delimiter character sequence separating the list items; `llDumpList2String([1,2,3,4,5], "--")` prints `"1--2--3--4--5"`. The function `llList2CSV()` is a specialized version, using only commas as separators.

TABLE 1.9: FUNCTIONS THAT CONVERT LISTS TO STRINGS

Function Name	Purpose
`string llDumpList2String(list source, string delimiter)`	Turns a list into a string, with the ***delimiter*** between items.
`list llParseString2List(string s, list delimiters, list spaces)`	Turns a string into a list, splitting at ***delimiters*** and ***spaces***, keeping ***spaces***. Consecutive ***delimiters*** are treated as one delimiter. Listing 1.4 shows an example of how to use this function.
`list llParseStringKeepNulls(string s, list delimiters, list spaces)`	Turns a string into a list; consecutive ***delimiters*** are treated separately.
`list llCSV2List(string csvString)`	Converts comma-separated values (CSVs) to a list. For instance, if ***csvString*** is a string "1,2,3" then the list will be [1, 2, 3].
`string llList2CSV(list l)`	Converts a list to a string containing comma-separated values (CSVs).

The `llParseString2List()` is particularly useful because it converts a string's elements to a list of strings. Listing 1.4 shows an example of how you would call it to break a string into separate items and then print them to the chat window.

Listing 1.4: `llParseString2List()` Example

```
default
{
    touch_start(integer total_number) {
        list l = llParseString2List("The answer to Life, the Universe"➥
                                    + " and Everything is 42", [" "], []);
        integer i;
        integer len = llGetListLength(l);
        for (i = 0; i < len; i++){
            llOwnerSay("item==>"+ llList2String(l, i));
        }
    }
}
```

While not totally symmetrical to `llDumpList2String()`, it still produces useful results:

```
[8:31]  Object: item==>The
[8:31]  Object: item==>answer
[8:31]  Object: item==>to
[8:31]  Object: item==>Life,
[8:31]  Object: item==>the
[8:31]  Object: item==>Universe
[8:31]  Object: item==>and
[8:31]  Object: item==>Everything
[8:31]  Object: item==>is
[8:31]  Object: item==>42
```

WARNING

If a list came from parsing a string using `llParseString2List()`, be aware that using `llList2<type>()` won't work. Instead you have to use an *explicit cast*, indicating that you want to change the type as in `(sometype)llList2String()`. The following snippet shows an example:

```
list components = llParseString2List(message,["|"],[]);
float gAngle = (float)llList2String(components,0);
vector gPosition = (vector)llList2String(components,1);
```

You can imitate objects or compound structures by using a ***strided list***, wherein the series of contained objects is repeated in a nonvarying sequence. The ***stride*** indicates how many elements are in each compound structure. Listing 1.5 creates a strided list, with a stride of 2, populating it with a string representation of the color name paired with the LSL vector of RGB values representing that color. It prints all the color names that occur between elements 2 and 4 of the list, namely green and blue.

Listing 1.5: Small Strided-List Example

```
list COLORS = ["red", <1., 0., 0.>, "green", <0., 1., 0.>,
               "blue", <0., 0., 1.>, "yellow", <1., 1., 0.>,
               "purple", <1., 0., 1.>, "cyan", <1., 0., 1.>];
integer STRIDE = 2;
default
{
    touch_start(integer total_number) {
        list keys = llList2ListStrided(COLORS, 2, 4, STRIDE);
        llOwnerSay(llDumpList2String(keys, ", "));
    }
}
```

There are some provisions for manipulating strided lists in LSL, including sorting and extracting elements such as keys, shown in Table 1.10. However, you will still need to have a few more utility functions. Put Listing 1.6 on your "save for later" pile—create the script and store it in your Inventory folder under Scripts. It contains a group of utility functions that enable the fetching, update, and deletion of a specific record.

TABLE 1.10: FUNCTIONS THAT MANIPULATE STRIDED LISTS

Function Name	Purpose
`list llList2ListStrided(list src, integer start, integer end, integer stride)`	Returns a list of all the entries whose index is a multiple of ***stride***, in the range of ***start*** to ***end*** inclusive.
`list llListSort(list l, integer stride, integer ascending)`	Sorts a list ascending (`TRUE`) or descending (`FALSE`), in blocks of length ***stride***. Works unreliably for heterogeneous lists.
`list llListRandomize(list l, integer stride)`	Randomizes a list in blocks of length ***stride***.

Listing 1.6: Managing Records in a Strided List

```
list gColors = ["red", <1.0, 0.0, 0.0>,
                "green", <0.0, 1.0, 0.0>,
                "blue", <0.0, 0.0, 1.0>];
integer STRIDE = 2;

integer getPosition(integer recordNumber)
{
    return recordNumber * STRIDE;
}
list getRecord(integer recordNumber)
{
    integer pos = getPosition(recordNumber);
    if (pos < (llGetListLength(gColors) + STRIDE - 1))
        return llList2List(gColors, pos, pos + STRIDE - 1);
    else
        return [];
}
deleteRecord(integer recordNumber)
{
    integer pos = getPosition(recordNumber);
    if (pos < (llGetListLength(gColors) + STRIDE - 1)) {
        gColors = llDeleteSubList(gColors, pos, pos + STRIDE - 1);
    }
}
updateRecord(integer recordNumber, list newRecord)
{
    integer pos = getPosition(recordNumber);
    if (pos < (llGetListLength(gColors) + STRIDE - 1)) {
        gColors = llListReplaceList(gColors, newRecord, pos,
                        pos + STRIDE - 1);
    }
}
default
{
    // embed some test code in a touch_start event handler
    // get record # 2
    touch_start(integer total_number) {
        integer i = 0;
        list l = getRecord(2);
        llOwnerSay(llDumpList2String(l, " + "));
    }
}
```

VARIABLES

Variables provide a place to store temporary working values. Variables can be both ***global*** and ***local***. Global variables are available to everything in the script, while local variables are available only to the block they were defined in. A variable name must start with an ASCII letter or an underscore. Numerals may be part of a variable name after the first character, and non-ASCII letters, while allowed, are ***ignored***. Thus, valid declarations include the following:

```
integer x;
integer y1;
integer _z;
integer thisIs_ALongComplicatedVariable_42;
integer enumerator;
integer _enumerator;
integer énumérateur;  // same as numrateur
```

However, you can not ***also*** declare an `integer` ***`numrateur`*** because LSL ignores the ***é***, and thus thinks the name has previously been declared in scope; that is, ***`numrateur`*** is the same variable as ***`énumérateur`***.

Global variables (and constants) must be defined before the `default` state, as shown in the following code:

```
integer gFoo = 42;
default
{
    on_rez(integer start_param) {
        llOwnerSay("gFoo is "+(string)gFoo);
    }
}
```

Variables can ***not*** be defined directly inside states (they must appear inside blocks, functions, or event handlers). Thus, a slight rearrangement of the preceding code will generate a syntax error:

```
default
{
    integer gFoo = 42;
    on_rez(integer start_param) {
        llOwnerSay("gFoo is "+(string)gFoo);
    }
}
```

Local variables must always be declared within the ***scope*** (enclosing curly braces) of a function or event handler but needn't be declared before the first executable statement. Just declare it prior to its first use and you'll be fine. Consider the following example:

```
foo() {
    integer bar = 0;
    llOwnerSay("bar is "+(string)bar);
    integer baz = 0;
    llOwnerSay("baz is "+(string)baz);
}
```

Using a local variable before it has been declared will generate the error "Name not defined in scope." Variables are scoped to the closest enclosing code block. Variables are not accessible outside this scope. After the function `foo()` returns, the variables ***bar*** and ***baz*** are unavailable. Thus, in the following code snippet the variable ***baz*** is not available outside the `else` branch of the `if (gBoolean)` test:

```
integer gBoolean = TRUE;
foo() {
    string bar = "Ford Prefect";
    if (gBoolean) {
        string bar = "Slarty Bartfast";
        llOwnerSay(bar);
    } else {
        string baz = "Zaphod Beeblebrox";
        llOwnerSay(baz);
    }
}
```

The snippet also demonstrates that you can ***shadow*** a variable from an outer code block with a redefinition of the same name. The innermost wins; that's why in this code example, `llOwnerSay(bar)` will always tell you "Slarty Bartfast" and ignore poor "Ford Prefect."

It is also useful to know that global variables can be ***set*** to the value returned by a function, but they can not be ***intialized*** to the value of a function. Similarly, LSL does not allow any code evaluation outside blocks, generating a syntax error for the following snippet:

```
float gHalfPi = PI/2;
default
{
}
```

Flow control is the process of directing the computer through your program in ways other than simply executing each line one after the other. If you have some experience with other computer languages, LSL flow control will come naturally to you, as most of the familiar constructs are here.

Conditionals are represented with `if`...`else` statements: they allow the ***conditional*** execution of code depending on the value of the expression following the word `if`:

- `if` (*condition*) *trueBranch*
- `if` (*condition*) *trueBranch* `else` *falseBranch*

The branches can be single statements; block statements (enclosed in braces, {***block***}); or null statements (empty or just a semicolon "`;`"). There is no LSL equivalent to the `switch` or `case` statements found in many other languages.

LSL provides a standard set of loop statements as well: `for`, `do...while`, and `while`:

- `do` { *loop* } `while` (*condition*) `;`
- `for` (*initializer*`;` *condition*`;` *increment*) { *loop* }
- `while` (*condition*) { *loop* }

Loop statements may also be single statements, block statements, or the empty statement ";". Loops may not declare variables in the initializer. That is, the following snippet is ***not*** allowed:

```
for (integer i=0; i<10; i++) {
    // loop
}
```

You can declare variables inside a loop block:

```
integer i;
for (i=0; i<10; i++) {
    float f=3.0;
    llOwnerSay((string)i+". "+(string)f);
}
```

A `do...while` loop is slightly faster than a `while` or `for` loop, and also requires fewer bytes of memory.

The flow statement `state` is described in the "States" section later in this chapter. `return` is used to return a value from a function, described in the section "User-Defined Functions." `jump` is just like the `goto` statement of many other languages, allowing direct jumping around the code of your script, and should generally be avoided. It is not uncommon in LSL for `jump` to be used to break out of loops:

```
integer i;
integer bigNumber = 1000;
for (i=0; i<bigNumber; i++) {
    if (weFoundOurNumber(i)) {
        jump doneWithIt;
    }
}
@doneWithIt;
llOwnerSay("Done after "+(string)i+"steps");
```

OPERATORS

Mathematical operators are the usual suspects (arithmetic, comparison, and assignment), with approximately the same precedence as you'd expect, such as multiplication and division before addition and subtraction. The exclamation point, `!`, denotes negation (returns `TRUE` for a `FALSE` value and vice versa). The tilde, ~, is the bitwise complement, flipping each bit of the integer it is applied to. The plus sign, +, can be used for a variety of things, including addition, string concatenation, and list appending.

NOTE

Unlike in most modern computer languages, Boolean operators in LSL do not "short-circuit" the calculation of an expression. If you were expecting short-circuiting, your code can have unexpected inefficiencies or broken logic.

Table 1.11 shows the operators, in approximate order of precedence. Empty cells will generate a compiler error with a type mismatch. Lists and strings are not in this table because they only support the addition operators + and +=, indicating concatenation. Note specifically that lists may not be compared with any operator, while strings can be compared with only == and != (but they compare only list ***length***, not list ***contents***). Table 1.12 lists the math-related library functions.

WARNING

There have been reports of unexpected (and unexplainable) precedence calculations in the expression parser, so use parentheses liberally.

TABLE 1.11. OPERATORS AND THEIR SEMANTICS FOR DIFFERENT TYPES

Operator	Integer (and Boolean)	Float	Vector	Rotation
`.`			Dot	Dot
`!`	Not			
`~`	Bitwise complement	Bitwise complement		
`++ --`	Increment, decrement	Increment, decrement		
`* / %`	Multiply, divide, modulus	Multiply, divide, modulus	Dot product, N/A, cross product	Addition, subtraction, N/A
`+ -`	Add, subtract	Add, subtract	Add, subtract	Legal, but semantically unlikely
`<< >>`	Left / right shift			
`< <= > >=`	Less than, less than or equal to, greater than, greater than or equal to	Less than, less than or equal to, greater than, greater than or equal to		
`== !=`	Comparison equal, comparison not equal	Comparison equal, comparison not equal		
`& ^ |`	Bitwise AND, XOR, OR			
`|| &&`	Comparison OR, AND			
`= += -= *= /= %=`	Assignment, with above semantics	Assignment	Assignment	Assignment

TABLE 1.12: LSL MATH FUNCTIONS

Function	Behavior
`integer llAbs(integer val)`	Returns an integer that is the positive version of the value ***val***.
`float llFabs(float val)`	Returns a float that is the positive version of ***val***.
`integer llRound(float val)`	Returns an integer that is ***val*** rounded to the closest integer. 0.0 to 0.499 are rounded down; 0.5 to 0.999 are rounded up.
`integer llCeil(float val)`	Returns the closest integer larger than ***val***.
`integer llFloor(float val)`	Returns the closest integer smaller than ***val***.

Function	Behavior
`float llFrand(float max)`	Returns a float that is a pseudorandom number in the range [0.0, ***max***) or (***max***, 0.0] if ***max*** is negative. That is, it might return 0.0 or any number up to but not including ***max***.
`float llSqrt(float val)`	Returns a float that is the square root of ***val***. Triggers a Math Error for imaginary results (***val*** < 0.0).
`float llLog(float val)`	Returns a float that is the natural logarithm of ***val***. If ***val*** `<= 0` it returns 0.0 instead.
`float llLog10(float val)`	Returns a float that is the base-10 logarithm of ***val***. If ***val*** `<= 0` it returns zero instead.
`float llPow(float base, float exp)`	Returns a float that is ***base*** raised to the power ***exp***.
`integer llModPow(integer base, integer exp, integer mod)`	Returns an integer that is ***base*** raised to the power ***exp***, modulo ***mod***: (***base***^{exp})% ***mod***). Causes the script to sleep for 1.0 seconds.
`float llSin(float theta)`	Returns a float that is the sine of ***theta*** (in radians).
`float llAsin(float val)`	Returns a float (***theta***, in radians) that is the inverse sine in radians of ***val***; that is, `sin(`***theta***`)` =***val***. ***val*** must fall in the range [–1.0, 1.0].
`float llCos(float theta)`	Returns a float that is the cosine of ***theta*** (in radians).
`float llAcos(float val)`	Returns a float (***theta***, in radians) that is the inverse cosine of ***val***; that is, `cos(`***theta***`)` =***val***. ***val*** must fall in the range [–1.0, 1.0].
`float llTan(float theta)`	Returns a float that is the tangent of ***theta*** (in radians).
`float llAtan2(float y, float x)`	Returns a float (***theta***, in radians) that is the arctangent of ***x***, ***y***. Similar to `tan(`***theta***`)` =***y***/***x***, except it utilizes the signs of ***x*** and ***y*** to determine the quadrant. Returns zero if ***x*** and ***y*** are zero.

FUNCTIONS

A ***function*** is a portion of code that performs a specific task. LSL supports two kinds of functions: ***built-in functions*** that are provided by Linden Lab, and ***user-defined functions*** that exist inside your scripts.

LSL includes a large number of built-in functions that extend the language (such as the ones you've already seen for math and list-manipulation functions). Built-in functions are readily identifiable as they always begin with the letters `ll`. Thus, you can tell immediately that `llSetTexture()` and `llOwnerSay()` are not user-defined functions. This book describes most of these Linden Library functions along with examples of how to use many of them.

User-defined functions are blocks of code that help modularize your code and improve readability. If you need to do the same thing more than once in your script, there's a good chance that code should be put into a function. User-defined functions must be defined before the `default` state. Functions are global in scope, and are available to all states, event handlers, and other user-defined functions in the same script (but not other scripts in an object). An example of the general form a function takes is shown in Listing 1.7.

Listing 1.7: Function Form

COMMENTS CONSTANTS VARIABLES TYPES LIBRARY FUNCTIONS KEYWORDS FUNCTION NAMES

```
// say something a few times
// return how many times message is said
integer saySomeTimes (string message) {
    integer count = 1 + (integer) llFrand(5.0);  // up to 5 times
    integer i;
    for ( i=0; i<count; i++ ) {
        llOwnerSay(message);
    }
    return count;
}
```

This function prints the string passed into it, ***message***, between one and five times to the owner's chat window.

A function has a name (`saySomeTimes` in Listing 1.7) and can have any number of parameters, each with a declared type (***message*** is a `string`). A function with no parameters is just declared as its name with an open/close parenthesis pair, as in `foo()`. Only parameter values are passed into the function, and the original value (in the caller's scope) is guaranteed to be unchanged. There is no concept of pass-by-reference in LSL. If you have large data structures, it's probably best to keep them as global variables rather than passing them as parameters into functions, because the entire data structure will be copied—this process takes time and memory.

Variables defined inside the function (***count*** and ***i***) have scope only inside the function block: nothing outside the function can see the variables. Be careful about naming your variables, especially if you use similar names in different scopes. It is remarkably easy to get confused about which variable is which when you reuse names.

If a function has a return type, it can return a single value of that type using the `return` statement. Functions that don't return a value should simply not declare a return type. The function `saySomeTimes()` in Listing 1.7 returns an integer of the number of times it chatted the message.

If you look back at Listing 1.2, you'll see the definition of the `listen()` event handler follows exactly the same form. In fact, an event handler is just a special function that is called by ***SL*** when certain events occur. Event handlers must be one of the event names known to ***SL***, must be declared to have the correct type parameters (but you may choose the names of the parameters!), and may not return values. The SYW website lists all the events, and there is more detail in the "Events" section of this chapter.

EVENTS AND EVENT HANDLERS

LSL is an ***event-driven*** language. This behavior is a major discriminating feature between LSL and many other languages. Essentially, the script's flow is determined by ***events*** such as receiving messages from other objects or avatars, or receiving sensor signals. Functionally, events are messages that are sent by a ***Second Life*** server to the scripts active in that sim. Event messages can be received by defining the matching ***event handlers*** in the state(s) of your scripts. ***Nothing*** happens in a script without event handlers: even the passing of time is marked by the delivery of timer events. Some events cannot happen without the matching event handler active somewhere in the object (in at least one of its prims). For instance, if an object doesn't have a script with a `money()` event handler, avatars cannot pay that object.

Many library functions have effects that take relatively long periods of time to accomplish. Rather than ***blocking*** (stopping execution), LSL uses ***asynchronous*** communications: your script fires off a request that something happen, and it is notified with an event when the request is satisfied. This pattern is used any time your script makes a call to a library function that cannot be handled locally by the host sim (for instance, the function `llHTTPRequest()` and the `http_response()` event handler), when physical simulation effects aren't going to be instantaneous (`llTarget()` and `at_target()`). You could also think of repeating events (`llSensorRepeat()` and `sensor()`) or even listeners (`llListen()` and `listen()`) as following the same model: make a request and get responses later. This model of asynchronous execution allows your script to keep functioning, handling other events and interactions without stopping to wait for long-term requests to happen.

As events occur, applicable ones (those that have handlers defined on the current state) are queued for execution by the script in the order they were raised. You should be aware of several important constraints on the delivery of event messages to handlers:

- By default, events are delivered to a script no more frequently than every 0.1 second. You can adjust this with `llMinEventDelay(float seconds)` but it can not be less than 0.05 second.
- A maximum of 64 events can be queued on a script—any additional events will be silently discarded!
- Queued events are silently discarded when a script transitions between states.

The combination of these factors means that if you have code that expects to get events rapidly and takes a relatively long time to execute (including artificial delays!), you may run the risk of missing events. As an example, sending an IM (which delays the script by 2 seconds) in response to collide events (which can happen as often as every 0.1 second) is probably asking for trouble.

Similarly, the LSL function `llSleep(float seconds)` pauses script execution for the specified number of seconds without leaving the current event handler, similar to the way that many LSL functions introduce artificial delays. In both cases, there is a potential problem if your script is likely to handle lots of events.

Some events are delivered to a script only if the script has made the appropriate request. For instance, a `listen()` event handler is never called unless another handler in the same state has called `llListen()`. The SYW website has a list of all the LSL event handlers, the functions required to enable the event, and the ***Second Life*** situation that results in that raised event. And, of course, there are lots of examples throughout the book.

STATES

Second Life models scripts as ***event-driven finite state machines*** similar to paradigms often used to run hardware. Wikipedia defines an event-driven finite state machine as "a model of behavior composed of a finite number of states, transitions between those states, and actions"* where actions occur as a result of the observation of externally defined events. This sort of model is often used in real-world applications where entirely predictable behavior is required, especially when programs are interacting directly with hardware or external events (think automotive electronics, traffic signals, and avionics).

This model is certainly apt for ***SL*** because many LSL scripts control virtual simulations of real-world mechanisms. Perhaps more importantly, it is useful as a simulator language because of how gracefully such programs scale under simulator load: under conditions of high "simulator lag," slowed event delivery might hurt performance, but nothing ought to break outright.

LSL scripts must implement at least the `default` state: indeed, many only use the `default` state, either exhibiting behavior that requires simple event-driven programming, or using global variables in place of states. A script will always be in exactly one state. All script execution occurs in the context of the current state, and in the current event. States are defined in an LSL script as sets of event handlers, labeled with the key words `default` or `state` *`statename`*.

Second Life calls two special event handlers, `state_entry()` and `state_exit()`, when your script changes states.

EVENT DEFINITION

```
state_entry( ) { }
```

This event handler is invoked on any state transition and in the default state when a script first runs or after it has been reset. This event is *not* triggered every time the object is rezzed. The SYW website provides additional documentation.

EVENT DEFINITION

```
state_exit( ) { }
```

This event handler is invoked when the `state` command is used to transition to another state, before the new state is entered. It is *not* invoked by `llResetScript()`. The SYW website provides additional documentation.

* `http://en.wikipedia.org/wiki/Finite_state_machine`

A script transitions between states using a `state newStateName` statement. Note that transition statements are not allowed inside user functions. Listing 1.8 shows a complete script that does nothing until touched, and then transitions to the exploding state and back to the default state. Figure 1.2 shows how the script execution passes through the states and event handlers.

Listing 1.8: Example State Transitions

```
default {
    touch_end(integer count) {
        state exploding;
    }
    state_exit() {
        llOwnerSay("The fuse has been lit.");
    }
}
state exploding {
    state_entry() {
        llOwnerSay("Boom!");
        state default;
    }
    state_exit() {
        llOwnerSay("Ash is now falling.");
    }
}
```

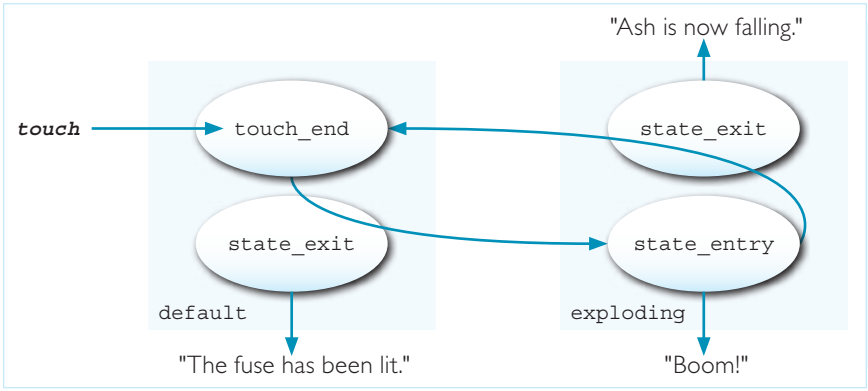

Figure 1.2: State transitions for Listing 1.9

A state transition will invoke the `state_exit()` event on the way out (still in the context of the origin state), allow it to run to completion, and then run the `state_entry()` event on the new state. In Listing 1.9 each state announces something as it is about to exit.

WARNING

Don't try state transitions from within a `state_exit()` event handler: it probably will not do what you hope for, and at the time of this writing it can result in some moderately bad (and varying) script behavior.

Additionally, avoid putting code after the command to transition states. It won't be executed, as shown in the following snippet:

```
touch_end( integer n ) {
    state exploding;
    llOwnerSay("I will never be executed!");
}
```

Any queued events, like touches or messages, that haven't been serviced yet are silently discarded during state transitions. Also, anything that was set up for delayed or repeated event handlers is invalidated: all in-progress listens, timers, sensors, and data server queries will need to be reopened. Global variable values survive state transitions, as well as granted permissions, taken controls, and XML-RPC channels.

WHEN TO USE MULTIPLE STATES (OR NOT!)

Multiple states are good when you want to strongly separate different modes of a scripted object. Consider a highly configurable object like a vendor kiosk: you ***really*** don't want people making purchases from the vendor while you are reconfiguring it. Even simple objects like a door or an elevator might best be represented with multiple states (***open*** versus ***closed***, and ***idle*** versus ***carrying*** passengers versus ***responding*** to a call, respectively). On the other hand, if you have a script where two states have a great deal of repeated code (event handlers) that cannot be abstracted out to functions, it may be simpler to collapse those two states together with a global variable. For instance, the default state from the texture-flipping script in Listing 1.2 could have been split into two using `default` as the "off" state and `lit` as the "on" state and eliminating the ***`gOn`*** variable, as shown in Listing 1.9.

Listing 1.9: Neon Texture Flipper as Multiple States

```
// Insert variables and appropriate functions from Listing 1.2
default
{
    state_entry() {
        fliptexture(NEON_OFF_TEXTURE);
        llSetTimerEvent(TIMER_INTERVAL);
    }
    timer() {
        state lit;
    }
}
state lit
{
    state_entry() {
        fliptexture(NEON_ON_TEXTURE);
    }
    timer() {
        state default;
    }
}
```

This snippet only sets the timer in the `default` state. Timers are special in that they are the only event requests not discarded when a script changes state. It would do no harm to re-invoke `llSetTimerEvent()` in its `state_entry()`. The SYW website discusses this in detail.

MANAGING SCRIPTED OBJECTS

The mechanics of dealing with scripted objects can be challenging at times. Scripts, after all, lend behavior to inanimate objects—and if your scripts aren't exactly right, misbehavior! Not to worry, though: while ***Second Life*** doesn't offer the sorts of software development and debugging tools that professional programmers have come to expect for serious work, there are techniques and tools that can help. For instance, Chapter 14, "Dealing with Problems," is all about finding and fixing problems. The SYW website features a compendium of resources for getting (and offering!) scripting help, as well as a survey of external LSL scripting tools. Furthermore, there are a few specific tips you may find useful as you start writing more-complicated scripts. Just keep in mind that bugs happen to everyone—be prepared for them and you'll be just fine.

LOSING MOVING OBJECTS

Normally you edit a prim or an object, create a script, and then edit the script. You can stop editing the prim and keep editing the script (assuming you leave the script window open). If the selected prim goes flying out of range into the sky, you will not be able to save changes until it is back in range. This is a pain when you lose track of the object (say, by rotating it to somewhere completely unexpected). There are some easy fixes to this problem: copy and paste the text of the old script into a new one, keep a copy in an out-of-world editor, or create a `timer()` event that returns the object to its original position, as in Listing 1.10.

Listing 1.10: Using a Timer to Reset an Object's Position and Orientation

```
vector gInitialPosition;
rotation gInitialOrientation;

default
{
    state_entry() {
        gInitialPosition = llGetPos();
        gInitialOrientation = llGetRot();
        llOwnerSay("Pos:"+(string)gInitialPosition);
        llOwnerSay("Rot:"+(string)gInitialOrientation);
    }
    touch_start(integer n) {
        llOwnerSay("Doing experimental stuff");
        // add your experiment here
        llSetTimerEvent( 10 );
    }
    timer() {
        llSetPos(gInitialPosition);
        llSetRot(gInitialOrientation);
        llSetTimerEvent( 0 );
    }
}
```

When the script starts the `state_entry()` event handler, it uses the functions `llGetPos()` and `llGetRot()` to find the position and rotation of the object and cache them. (These functions are described in Chapter 4.) When the user touches the object, the `touch_start()` handler runs the experimental code, and, before exiting, sets up a timer event with `llSetTimerEvent()`. Magically, 10 seconds later when the `timer()` event triggers, the object is moved back to its original position using `llSetPos()` and `llSetRot()`. Note that `llSetPos()` is limited to moves of 10m or less; Chapter 2 shows how to move the object farther.

DEFINITION

`llSetTimerEvent(float` ***`timerInterval`***`)`

Cause a `timer()` event to be triggered once every ***`timerInterval`*** seconds. A value of 0.0 stops further timer events.

`timerInterval` — Any positive nonzero value.

EVENT DEFINITION

`timer( ) { }`

This event handler is invoked when triggered at periodic intervals specified by the parameter to `llSetTimerEvent(float` ***`interval`***`)`. Only one timer can be active in the state at one time. The SYW website provides additional documentation.

A simple experiment to try would be to replace the commented line in the `touch_start()` event handler with the following snippet:

```
vector newPos = gInitialPosition + <1,1,1>;
llSetPos(newPos);
```

MULTIPLE SCRIPTS

A ***Second Life*** prim may contain any number of running scripts in its inventory and, of course, ***SL*** objects are often made from multiple prims. The result is that a complex scripted object can have many scripts cooperating to produce the object's behavior. Many scripters develop a suite of small, focused scripts to accomplish different tasks, rather like programmers in other languages often build libraries of utility functions. For example, you might develop a comfy pillow that has three scripts—one for sitting, one for changing textures, and one that is a sound trigger for some mood music if an avatar is sensed nearby. A suite of scripts like this can be combined to form new behaviors.

This modularity also makes it easier to debug, maintain, and reuse; store them in the Scripts section of your Inventory folder. Best of all, smaller scripts are more stable. Each script executes within its own block of memory, and ***can not*** accidentally write into the memory of other scripts or the protected simulator memory; this protection makes it much harder for your scripts to crash the simulator!

Multiple scripts on the same object run in parallel, as close to simultaneously as the sim server can manage, and are fired in no defined order. Each script can be in a different state, making for interesting possibilities. In Chapter 3 you'll see how to synchronize behavior among the scripts.

RESETTING SCRIPTS

Scripts reset to default values when they are placed in another object. However, if a scripted object is in your possession and you rez it or transfer ownership, the script is ***not*** reset. Scripted objects do not "instinctively" reset when transferred, and thus either an explicit reset or careful handling of the change of ownership is often required. `llResetScript()` forces a reset of the script, restarting it from scratch. All global variables are set to their defaults, all pending events are cleared, the `default` state becomes active, and its `state_entry()` is triggered. When writing and debugging the script, you can achieve an identical effect by clicking Reset in the script editor.

DEFINITION

`llResetScript()`

Resets the script completely, restarting it from scratch. This approach is a good way to catch ownership transfers. It does not correct persistent prim features such as sit text, a sit target, or floating text.

Certain persistent features of an object are ***not*** reset with `llResetScript()`, including floating text and sit targets; these must be explicitly deleted. Every script that maintains state must take care to reset when transferred, either by detecting transfers or resetting when rezzed. Our scripts do the latter by catching the `on_rez()` event, as shown in the following snippet. If you don't reset the script or double-check who the owner is, for example, a chatting script would still listen to the original owner.

```
default
{
    on_rez(integer start_param) {
        llResetScript();
    }
}
```

EVENT DEFINITION

`on_rez(integer startParam) { }`

This event handler is invoked when an object is rezzed into the world from inventory or by another script. The SYW website provides additional documentation.

startParam **— The parameter the prim starts with; zero for an object rezzed from inventory, and any value if rezzed by another script.**

You can also keep track of the expected owner in a global variable, and reset or adapt the script only when the owner changes:

```
key gOwner;
default
{
    state_entry() {
        gOwner = llGetOwner();
    }
    on_rez(integer p) {
        if (gOwner != llGetOwner()) {
            llResetScript();
        }
    }
}
```

WARNING

This book contains scripts that use `llResetScript()` in many different places depending on specific needs. It's usually needed in an `on_rez()` event handler; however, one place to *never* use it is in the `state_entry()` handler—it will result in infinite recursion.

Two LSL functions allow one script to control the state of another script in the same prim: `llSetScriptState(string statename, integer run)`, which causes a state transition in the current script to the named state, and `llResetOtherScript(string scriptname)`, which resets the named script.

AN LSL STYLE GUIDE

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*
>
> —Martin Fowler et al, *Refactoring: Improving the Design of Existing Code*

Effective programming in LSL requires that developers use a disciplined practice for applying formatting and convention to their scripts. These guidelines are not as rigid as the rules required by the language compiler, but nonetheless are critical to creating maintainable code. The most critical aspect of a style is that you apply it consistently to the code you write. We are attempting to make a guideline, not a rigid coding style, and thus point out things that ***must*** be followed to make it compile and things that ***should*** be followed to make your scripts readable. The SYW website describes a variety of tools you can use outside of ***SL*** to help you script, but the principles are important no matter how you write your scripts.

CLEANLINESS

The most important practice in effective programming is to generate readable code. Readable code is easier to debug, understand, and maintain. A short snippet that is poorly formatted may be decipherable later, but a longer program will not be. If you pay attention to your coding style early on, it will quickly become second nature.

Here is a list of specific suggestions:

- ***Bracketing***: You must always enclose blocks in curly braces. It is a good idea to always use braces, even for single statements. Opening braces for functions and states should start on the line after the declaration. Otherwise, place the brace on the same line as the control block.
- ***Indentation***: On the line immediately following an open brace, you should add one indentation, usually four spaces. Remove an indentation on the line after a close brace.

NOTE

The two most important guidelines are indentation and bracketing. This code, for example, is not properly indented:

```
default{state_entry(){ llSay(0, "Hello, Avatar!"); } touch_
    start(integer total_number)  {  llSay(0, "Touched."); }}
```

If you look closely, you'll see that it's the same as Listing 1.1—which was much more readable!

- ***Line wrap***: You should manually separate lines longer than 80 characters.
- ***Coding styles***: In general, don't mix coding styles. When editing code you didn't create, keep the existing style (or change the style throughout).
- ***Function length***: Try to keep functions short and focused. A good rule is that a function longer than a screen height should be refactored into multiple functions.
- ***Comments***: Use comments to explain what you are doing; however, avoid excessive inline comments: well-written code should explain itself. Do, however, describe what each function does in a comment near the top of the definition.
- ***Variable naming***:
  - Variable names must always start with a letter or underscore.
  - Don't abbreviate. Since case matters (`x` is not the same as `X`), longer names will cause significantly less confusion.
  - Global variables should begin with a lower case **g**; for example, ***`gSelected`***.
  - Variables used as constants should be in all caps, and words should be separated by underscores, like this: `OWNER_KEY`.
  - You should use camel case (lowercase first letter, and then capitalize each word) for variable names, such as this: ***`myCountingVariable`***.

- *Function naming:*
  - Function names must also start with a letter, and usually are camel case, with each word capitalized similarly to the Linden Library functions like `doSomethingInteresting()`.
  - Linden Lab uses the prefix `ll` for built-in functions, such as `llGetScriptName()`, so don't use this syntax for your own code—it is considered very bad form. Look for `llPizza()` on the LSL wiki; it has a short discussion of user functions masquerading as Linden Library functions.
  - If you are creating functions that belong to a family, and you are likely to reuse them regularly, you should define a consistent naming convention across the family. LSL does not have an explicit "library" concept, but if you stick to one convention, you will find it easier to maintain a collection of reusable code for your projects.

NOTE

To conserve space, the scripts in this book don't follow all the rules all the time, especially for comments. The prose around the listings should be sufficient explanation.

MODULARITY

LSL doesn't offer any tools for writing and maintaining modular or especially reusable scripts, however, here are some tricks:

- Write and edit your code outside ***Second Life***, only pulling it in for testing, integration into builds, and deployment. Not only does this approach allow you to use superior editors, but it gives you access to revision-control systems and code generators and transformers—see the SYW website for more.
- Package separable parts as different scripts in the same object. This can keep the individual pieces smaller if the parts don't require close coordination.
- Use ***worker scripts*** to do slow (artificially delayed) tasks, or jobs that require one script per avatar. This will usually require a "brain" script that instructs a set of subsidiary scripts what to do, so they can act in concert so that the brain isn't delayed while waiting for something to happen.

SUMMARY

This chapter should have given you a good understanding of the LSL language and the basics of how to use it to manipulate ***Second Life*** functions. You should be aware that LSL is an evolving language; Linden Lab is fixing bugs and adding new capabilities all the time. That means you may run into something that isn't working the way you expect. Before it drives you up a tree, check the ***Second Life*** website of formal issues: `http://jira.secondlife.com`. The SYW website also has tips that can help get you out of a jam.

Most of the rest of this book is devoted to making effective use of LSL and the library to accomplish interesting things in ***SL***. After you've absorbed some of the ideas elsewhere in the book, come back and re-read this chapter. Even for us, the authors, re-reading has been beneficial. We wrote this chapter, then spent several weeks doing heavy-duty scripting before the editors were ready for us to make a second pass on Chapter 1. Our re-read was enlightening, helping us understand certain things better or in a different way than we did when we first wrote the text. Take it from us: there are lots of things that will make much more sense on later perusal.

And now, on to the fun stuff!